

SIGMOD Officers, Committees, and Awardees

Chair	Vice-Chair	Secretary/Treasurer
Angela Bonifati LIRIS - Département Informatique Université Claude Bernard Lyon 1 Lyon France +33-4-72-44-58-24 Angela.bonifati <at> univ-lyon1.fr	Alexandra Meliou College of Information and Computer Science University of Massachusetts Amherst, Massachusetts USA +1-413-545-3788 ameli <at> cs.umass.edu	Manos Athanassoulis College of Arts and Sciences Boston University Boston, Massachusetts USA +1-617-353-4602 mathan <at> bu.edu

SIGMOD Executive Committee:

Angela Bonifati (Chair), Alexandra Meliou (Vice-chair), Manos Athanassoulis (Treasurer), Divyakant Agrawal (Previous SIGMOD Chair), Chris Jermaine (SIGMOD Conference Coordinator), Rada Chirkova (SIGMOD Record Editor), Joe Hellerstein (2025 SIGMOD PC co-chair), Azza Abouzied (2025 SIGMOD PC co-chair), Floris Geerts (Chair of PODS), Genoveva Vargas Solar (SIGMOD Diversity and Inclusion Coordinator), Sourav S Bhowmick (SIGMOD Ethics), Byron Choi (SIGMOD Information Director), Yufei Tao (ACM TODS Editor in Chief)

Advisory Board:

Yannis Ioannidis (Chair), Phil Bernstein, Surajit Chaudhuri, Rakesh Agrawal, Joe Hellerstein, Mike Franklin, Laura Haas, Renee Miller, John Wilkes, Chris Olsten, AnHai Doan, Tamer Özsu, Gerhard Weikum, Stefano Ceri, Beng Chin Ooi, Timos Sellis, Sunita Sarawagi, Stratos Idreos, and Tim Kraska

SIGMOD Information Directors:

Sourav S Bhowmick, Nanyang Technological University
Byron Choi, Hong Kong Baptist University

Associate Information Directors:

Hui Li (SIGMOD Record), Georgia Koutrika (Blogging), Wim Martens (PODS)

SIGMOD Record Editor-in-Chief:

Rada Chirkova, NC State University

SIGMOD Record Associate Editors:

Lyublena Antova, Manos Athanassoulis, Angela Bonifati, Renata Borovica-Gajic, Vanessa Braganholo, Aaron J. Elmore, George Fletcher, Wook-Shin Han, H V Jagadish, Alfons Kemper, Benny Kimelfeld, Samuel Madden, Kyriakos Mouratidis, Tamer Özsu, Kenneth Ross, Pinar Tözün, Immanuel Trummer, Yannis Velegrakis, and Ke Yi

SIGMOD Conference Coordinator:

Chris Jermaine, Rice University

PODS Executive Committee:

Floris Geerts (chair), Pablo Barcelo, Leonid Libkin, Hung Q. Ngo, Reinhard Pichler, Dan Suciu

Sister Society Liaisons:

Raghu Ramakrishnan (SIGKDD), Yannis Ioannidis (EDBT Endowment), Christian Jensen (IEEE TKDE)

SIGMOD Awards Committee:

Sharad Mehrotra (Chair), H.V. Jagadish, Sourav S Bhowmick, Angela Bonifati, David Maier, Rachel Pottinger, Sayan Ranu, Cyrus Shahabi, Wang-Chiew Tan

Jim Gray Doctoral Dissertation Award Committee:

Evaggelia Pitoura (chair), Angela Bonifati (co-chair), Sourav S Bhowmick, Daniel Kang, Georgia Koutrika, Supun Nakandala, Fatma Ozcan, Julia Stoyanovich, and Xiaofang Zhou

SIGMOD Edgar F. Codd Innovations Award

For innovative and highly significant contributions of enduring value to the development, understanding, or use of database systems and databases. Recipients of the award are the following:

Michael Stonebraker (1992)	Jim Gray (1993)	Philip Bernstein (1994)
David DeWitt (1995)	C. Mohan (1996)	David Maier (1997)
Serge Abiteboul (1998)	Hector Garcia-Molina (1999)	Rakesh Agrawal (2000)
Rudolf Bayer (2001)	Patricia Selinger (2002)	Don Chamberlin (2003)
Ronald Fagin (2004)	Michael Carey (2005)	Jeffrey D. Ullman (2006)
Jennifer Widom (2007)	Moshe Y. Vardi (2008)	Masaru Kitsuregawa (2009)
Umeshwar Dayal (2010)	Surajit Chaudhuri (2011)	Bruce Lindsay (2012)
Stefano Ceri (2013)	Martin Kersten (2014)	Laura Haas (2015)
Gerhard Weikum (2016)	Goetz Graefe (2017)	Raghu Ramakrishnan (2018)
Anastasia Ailamaki (2019)	Beng Chin Ooi (2020)	Alon Halevy (2021)
Dan Suciu (2022)	Joseph M. Hellerstein (2023)	Samuel Madden (2024)
Carlo Zaniolo (2025)		

SIGMOD Systems Award

For technical contributions that have had significant impact on the theory or practice of large-scale data management systems.

Michael Stonebraker and Lawrence Rowe (2015); Martin Kersten (2016); Richard Hipp (2017); Jeff Hammerbacher, Ashish Thusoo, Joydeep Sen Sarma; Christopher Olston, Benjamin Reed, and Utkarsh Srivastava (2018); Xiaofeng Bao, Charlie Bell, Murali Brahmadesam, James Corey, Neal Fachan, Raju Gulabani, Anurag Gupta, Kamal Gupta, James Hamilton, Andy Jassy, Tengiz Kharatishvili, Sailesh Krishnamurthy, Yan Leshinsky, Lon Lundgren, Pradeep Madhavarapu, Sandor Maurice, Grant McAlister, Sam McKelvie, Raman Mittal, Debanjan Saha, Swami Sivasubramanian, Stefano Stefani, and Alex Verbitski (2019); Don Anderson, Keith Bostic, Alan Bram, Grg Burd, Michael Cahill, Ron Cohen, Alex Gorrod, George Feinberg, Mark Hayes, Charles Lamb, Linda Lee, Susan LoVerso, John Merrells, Mike Olson, Carol Sandstrom, Steve Sarette, David Schacter, David Segleau, Mario Seltzer, and Mike Ubell (2020); Michael Blanton, Adam Bolton, Bill Boroski, Joel Brownstein, Robert Brunner, Tamas Budavari, Sam Carliles, Jim Gray, Steve Kent, Peter Kunszt, Gerard Lemson, Nolan Li, Dmitry Medvedev, Jeff Munn, Deoyani Nandrekhar-Heinis, Maria Nieto-Santisteban, Wil O'Mullane, Victor Paul, Don Slutz, Alex Szalay, Gyula Szokoly, Manu Taghizadeh-Popp, Jordan Raddick, Bonnie Souter, Ani Thakar, Jan Vandenberg, Benjamin Alan Weaver, Anne-Marie Weijmans, Sue Werner, Brian Yanny, Donald York, and the SDSS collaboration (2021); Michael Armbrust, Tathagata Das, Ankur Dave, Wenchen Fan, Michael J. Franklin, Huaxin Gao, Maxim Gekk, Ali Ghodsi, Joseph Gonzalez, Liang-Chi Hsieh, Dongjoon Hyun, Hyukjin Kwon, Xiao Li, Cheng Lian, Yanbo Liang, Xiangrui Meng, Sean Owen, Josh Rosen, Kousuke Saruta, Scott Shenker, Ion Stoica, Takuya Ueshin, Shivaram Venkataraman, Gengliang Wang, Yuming Wang, Patrick Wendell, Reynold Xin, Takeshi Yamamuro, Kent Yao, Matei Zaharia, Ruifeng Zheng, and Shixiong Zhu (2022); Aljoscha Krettek, Andrey Zagrebin, Anton Kalashnikov, Arvid Heise, Asterios Katsifodimos, Jiangji (Becket) Qin, Benchao Li, Bowen Li, Caizhi Weng, ChengXiang Li, Chesnay Schepler, Chiwan Park, Congxian Qiu, Daniel Warneke, Danny Cranmer, David Anderson, David Morávek, Dawid Wysakowicz, Dian Fu, Dong Lin, Eron Wright, Etienne Chauchot, Fabian Hueske, Fabian Paul, Feng Wang, Gabor Somogyi, Gary Yao, Godfrey He, Greg Hogan, Guowei Ma, Gyula Fora, Haohui Mai, Henry Saputra, Hequn Cheng, Igal Shilman, Ingo Bürk, Jamie Grier, Jark Wu, Jincheng Sun, Jing Ge, Jing Zhang, Jingsong Lee, Junhan Yang, Konstantin Knauf, Kostas Kloudas, Kostas Tzoumas, Kete (Kurt) Young, Leonard Xu, Lijie Wang, Lincoln Lee, Lungu Andra, Martijn Visser, Marton Balassi, Matthias J. Sax, Matthias Pohl, Matyas Orhidi, Maximilian Michels, Nico Kruber, Niels Basjes, Paris Carbone, Piotr Nowojski, Qingsheng Ren, Robert Metzger, Roman Khachatryan, Rong Rong, Rui Fan, Rui Li, Sebastian

Schelter, Seif Haridi, Sergey Nuyanzin, Seth Wiesman, Shaoxuan Wang, Shengkai Fang, Shuyi Chen, Sihua Zhou, Stefan Richter, Stephan Ewen, Theodore Vasiloudis, Thomas Weise, Till Rohrmann, Timo Walther, Tzu-Li (Gordon) Tai, Ufuk Celebi, Vasiliki Kalavri, Volker Markl, Wei Zhong, Weijie Guo, Xiaogang Shi, Xiaowei Jiang, Xingbo Huang, Xingcan Cui, Xintong Song, Yang Wang, Yangze Guo, Yingjie Cao, Yu Li, Yuan Mei, Yun Gao, Yun Tang, Yuxia Luo, Zhijiang Wang, Zhipeng Zhang, Zhu Zhu, Zili Chen (2023); Zhaojing Luo, Beng Chin Ooi, Wei Wang, Meihui Zhang, Qingchao Cai, Shaofeng Cai, Gang Chen, Tien Tuan Anh Dinh, Jinyang Gao, Qian Lin, Shicong Lin, Kee Yuan Ngiam, Gene Yan Ooi, Moaz Reyad, Kian-Lee Tan, Anthony K. H. Tung, Sheng Wang, Yuncheng Wu, Zhongle Xie, Naili Xing, Rulin Xing, Wanqi Xue, Sai Ho Yeung, James Yip, Lingze Zeng, Zhaoqi Zhang, Kaiping Zheng, Lei Zhu, Ji Wang (2024); James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Alexander Lloyd, Sergey Melnik, David Mwaura, Sean Quinlan, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, Dale Woodford, David F. Bacon, Shannon Bales, Nico Bruno, Brian F. Cooper, Adam Dickinson, Campbell Fraser, Milind Joshi, Eugene Kogan, Rajesh Rao, David Shue, Marcel van der Holst, Cliff Frey, Damian Reeves, Steve Middlekauff, Mert Akdere, Ben Vandiver, Dan Glick, David Ziegler, Alex Khesin, Dave Weissman, Todd Lipcon, Sean Dorward, Eric Veach (2025).

SIGMOD Contributions Award

For significant contributions to the field of database systems through research funding, education, and professional services. Recipients of the award are the following:

Maria Zemankova (1992)	Gio Wiederhold (1995)	Yahiko Kambayashi (1995)
Jeffrey Ullman (1996)	Avi Silberschatz (1997)	Won Kim (1998)
Raghu Ramakrishnan (1999)	Michael Carey (2000)	Laura Haas (2000)
Daniel Rosenkrantz (2001)	Richard Snodgrass (2002)	Michael Ley (2003)
Surajit Chaudhuri (2004)	Hongjun Lu (2005)	Tamer Özsu (2006)
Hans-Jörg Schek (2007)	Klaus R. Dittrich (2008)	Beng Chin Ooi (2009)
David Lomet (2010)	Gerhard Weikum (2011)	Marianne Winslett (2012)
H.V. Jagadish (2013)	Kyu-Young Whang (2014)	Curtis Dyreson (2015)
Samuel Madden (2016)	Yannis E. Ioannidis (2017)	Z. Meral Özsoyoğlu (2018)
Ahmed Elmagarmid (2019)	Philippe Bonnet (2020)	Juliana Freire (2020)
Stratos Idreos (2020)	Stefan Manegold (2020)	Ioana Manolescu (2020)
Dennis Shasha (2020)	Divesh Srivastava (2021)	Christian S. Jensen (2022)
K. Selcuk Candan (2023)	Sihe Amer-Yahia (2024)	
Hector Munoz-Avila & Sylvia Spengler (2025)		

SIGMOD Jim Gray Doctoral Dissertation Award

SIGMOD has established the annual SIGMOD Jim Gray Doctoral Dissertation Award to *recognize excellent research by doctoral candidates in the database field.* Recipients of the award are the following:

- **2006 Winner:** Gerome Miklau. *Honorable Mentions:* Marcelo Arenas and Yanlei Diao
- **2007 Winner:** Boon Thau Loo. *Honorable Mentions:* Xifeng Yan and Martin Theobald
- **2008 Winner:** Ariel Fuxman. *Honorable Mentions:* Cong Yu and Nilesh Dalvi
- **2009 Winner:** Daniel Abadi. *Honorable Mentions:* Bee-Chung Chen and Ashwin Machanavajjhala
- **2010 Winner:** Christopher Ré. *Honorable Mentions:* Soumyadeb Mitra and Fabian Suchanek
- **2011 Winner:** Stratos Idreos. *Honorable Mentions:* Todd Green and Karl Schnaitterz
- **2012 Winner:** Ryan Johnson. *Honorable Mention:* Bogdan Alexe
- **2013 Winner:** Sudipto Das, *Honorable Mention:* Herodotos Herodotou and Wenchao Zhou
- **2014 Winners:** Aditya Parameswaran and Andy Pavlo.
- **2015 Winner:** Alexander Thomson. *Honorable Mentions:* Marina Drosou and Karthik Ramachandra
- **2016 Winner:** Paris Koutris. *Honorable Mentions:* Pinar Tozun and Alvin Cheung
- **2017 Winner:** Peter Bailis. *Honorable Mention:* Immanuel Trummer
- **2018 Winner:** Viktor Leis. *Honorable Mention:* Luis Galárraga and Yongjoo Park
- **2019 Winner:** Joy Arulraj. *Honorable Mention:* Bas Ketsman
- **2020 Winner:** Jose Faleiro. *Honorable Mention:* Silu Huang
- **2021 Winner:** Huanchen Zhang, *Honorable Mentions:* Erfan Zamanian, Maximilian Schleich, and Natacha

Crooks

- **2022 Winner:** Chenggang Wu, *Honorable Mentions:* Pingcheng Ruan and Kexin Rong
- **2023 Winner:** Supun Nakandala, *Honorable Mentions:* Benjamin Hilprecht and Zongheng Yang
- **2024 Winner:** Daniel Kang, *Honorable Mentions:* Wei Dong, Jialin Ding, and Yisu Remy Wang
- **2025 Winner:** Peng Li, *Honorable Mentions:* Xuanhue Zhou, Aecio Santos, and Meghdad Kurmanji

A complete list of all SIGMOD Awards is available at: <https://sigmod.org/sigmod-awards/>

[Last updated: October 1, 2025]

Guest Editors' Notes

Welcome to the March 2026 issue of the ACM SIGMOD Record, a special issue presenting papers and technical perspectives covering a selection of the best conference papers from 2025 in data management. These papers have been given the 2025 ACM SIGMOD Research Highlight Award. This is an award for the database community to showcase a set of research projects that exemplify core database research. In particular, these projects address an important problem, the papers represent a definitive milestone in solving the problem, and have the potential of significant impact. This award also aims to make the selected works widely known in the database community, to our industry partners, and to the broader ACM community.

The award committee and editorial board included Carsten Binnig, Rada Chirkova, Alfons Kemper, Samuel Madden, Julia Stoyanovich, and Ke Yi (chair). We solicited articles from SIGMOD, PODS, VLDB, ICDE, EDBT, and ICDT, published in 2025, as well as from community nominations. We used a careful review process, in which each nominated paper was discussed in a virtual meeting. Papers with conflict of interest were discussed in the absence of the conflicting committee members. This year, ten articles were finally selected as 2025 Research Highlight Award winners.

The authors of each article worked closely with an associate editor to rewrite the article into a compact 8-page format and improved it to appeal to the broad data management community. In addition, each research highlight in the March 2026 issue is accompanied by a one-page technical perspective written by an expert on the topic presented in the article. The technical perspective provides the reader with an overview of the background, the motivation, and the key innovation of the featured research highlight, as well as its scientific and practical significance.

2025 ACM SIGMOD Research Highlight Award Winners

1. A theoretical breakthrough on the join order problem together with practical improvements (“DPconv: Super-Polynomially Faster Join Ordering”);
2. A technique to vectorize multiple instances of a vertex-centric algorithm (“Automating Vectorized Distributed Graph Computation”);
3. An optimal algorithm for computing conjunctive queries in terms of the output size (“Output-Optimal Algorithms for Join-Aggregate Queries”);
4. “Output-sensitive Conjunctive Query Evaluation”, which has overlapping results with the previous paper (a combined highlight paper is written by the authors of the two papers jointly);
5. A method to analyze collections of documents while preserving strong privacy guarantees (“Differentially Private Substring and Document Counting”);
6. A dynamic range filter that simultaneously supports updates, variable-length range queries, and variable-length keys (“Diva: Dynamic Range Filter for Var-Length Keys and Queries”);
7. A generic and self-decoding framework for easy integration of data formats and data system (“AnyBlox: A Framework for Self-Decoding Datasets”);

8. A new method for optimizing UDFs in SQL databases by combining UDF outlining with selective inlining (“The Key to Effective UDF Optimization: Before Inlining, First Perform Outlining”);
9. A framework for reusing intermediate results and memory allocations in multi-backend machine-learning systems (“MEMPHIS: Holistic Lineage-based Reuse and Memory Management for Multi-backend ML Systems”);
10. A new programming language for relational data that abandons the traditional sublanguage paradigm and instead constructs a language from a minimal core that can evolve through libraries (“Rel: A Programming Language for Relational Data”).

On behalf of the SIGMOD Record Editorial Board, we hope that you enjoy reading the March 2026 issue of the SIGMOD Record!

Carsten Binnig, Rada Chirkova, Alfons Kemper, Samuel Madden, Julia Stoyanovich, Ke Yi

March 2026

Technical Perspective: DPconv: Super-Polynomially Faster Join Ordering

Guido Moerkotte
University of Mannheim

On the floors of database conferences, one can often listen to database researchers talking about the join ordering problem. However, *the* join ordering problem does not exist! In fact, depending on the properties of the query graph, the cardinality estimation method, the cost function, and the resulting output join tree, there exist multiple join ordering problems. For example, if the query graph is connected and acyclic, the cardinality estimator uses the independence assumption, the cost function has the ASI property, and the resulting join tree is a left-deep tree without cross products, then the problem can be solved in polynomial time [3].

The join ordering problem discussed in the paper is specified by no restriction on the query graph (it can even have no edges), no restriction on the cardinality estimation method, and for the resulting join trees can be bushy with cross products. The framework DP_{conv} transforms the join ordering problem to some kind of subset space, solves the problem therein, and transforms the solution back; this is similar in spirit to discrete FFT. Thereby, DP_{conv} exhibits a reduced complexity compared to traditional ones working on the same join ordering problem (such as DP_{sub}): It is the first approach to beat the $O(3^n)$ complexity barrier.

On the cost function side, the authors instantiate their framework with two cost functions: $C_{\max}$ and C_{out} . A direct instantiation of the framework with $C_{\max}$ yields a complexity of $O(2^n n^3)$, where a factor of n (the number of relations) can be saved by a simpler algorithm. Thus, from a complexity point of view things look great. From a practicability point of view, assume a query whose result size is at least the size of any intermediate result (e.g., cross products only). Then, *every* plan under $C_{\max}$ has the same cost. In general, since the number of possible plans $\binom{2(n-1)}{(n-1)}$ by far exceeds the number of possibilities for $C_{\max}$ ($2^n - 1$), many plans will end up with the same costs. Consequently, we believe that a direct application of $C_{\max}$ for plan generation may fail. Nonetheless, the authors point to another interesting practical application of $C_{\max}$: use its result to prune the search space. To this end, they present some experiments using a pruning version of DP_{sub}. Since top-down algorithms like TD_{mcc} [1] are typically better suited for pruning, it will be interesting to see how effective this pruning measure will be compared to and combined with existing ones.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 2025 ACM 0001-0782/24/0X00 ...\$5.00.

The instantiation of DP_{conv} with C_{out} results in a complexity of $O(2^n n^2 W n \log W n)$, where W is the cardinality of the largest intermediate result. For query 29c of the join order benchmark [4], the factor $W \log W$ alone is more than 1000 times larger than 3^n , the complexity of DP_{sub}. If we do not consider cross products, the factor $W \log W$ is already about 500K times larger than the number of ccps. This renders the approach infeasible for practice. However, remember that C_{out} was originally proposed to study the complexity of different join ordering problems. Only later it turned out that it is a good proxy for real costs whose application results in join orders which are quite good. Thus, an important line of future work opens up: Can we define properties of cost functions and families of cost functions satisfying (some of) these properties such that (a) DP_{conv} allows for an efficient instantiation and (b) the cost functions are good proxies for real costs if applied to join ordering? Another alternative to improve the efficiency of DP_{conv} would be to take the query graph into account by excluding cross products. This would require some progress in sparse subset convolution.

Parallelizing join ordering is another important topic. The candidate of choice has thus far been DP_{sub} [2, 5]. Now, DP_{conv} offers an alternative. A closer look at DP_{conv} reveals that it is very well suited for massive parallelization. Hence, I look forward to a first implementation and evaluation of DP_{conv} on a GPU.

1. REFERENCES

- [1] P. Fender and G. Moerkotte. A new, highly efficient, and easy to implement top-down join enumeration algorithm. In *Proc. IEEE Conference on Data Engineering*, pages 864–875, 2011.
- [2] W. Han, W. Kwak, J. Lee, G. Lohman, and V. Markl. Parallelizing query optimization. *Proc. of the VLDB Endowment (PVLDB)*, 1(1):188–200, 2008.
- [3] T. Ibaraki and T. Kameda. Optimal nesting for computing n-relational joins. *ACM Trans. on Database Systems*, 9(3):482–502, 1984.
- [4] V. Leis, B. Radke, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann. Query optimization through the looking glass, and what we found running the join order benchmark. *VLDB Journal*, 27:643–668, 2018.
- [5] R. Mancini, S. Karthik, B. Chandra, V. Mageirakos, and A. Ailamaki. Efficient massively parallel join optimization for large queries. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, 2022.

DPconv: Super-Polynomially Faster Join Ordering

Mihail Stoian
University of Technology Nuremberg
mihail.stoian@utn.de

Andreas Kipf
University of Technology Nuremberg
andreas.kipf@utn.de

ABSTRACT

We revisit the join ordering problem in query optimization. The standard exact algorithm, **DPccp**, has a worst-case running time of $O(3^n)$. This is prohibitively expensive for large queries, which are not that uncommon anymore. We develop a new algorithmic framework based on subset convolution. **DPconv** achieves a super-polynomial speedup over **DPccp**, breaking the $O(3^n)$ time-barrier for the first time. We show that the framework instantiation for the $C_{\max}$ cost function is up to 30x faster than **DPccp** for large clique queries.

1. INTRODUCTION

The query optimizer is the heart of any relational database system. One of the fundamental tasks of the query optimizer is join ordering. The problem is to reorder the joins, so that the query execution time is minimized. To this end, one introduces a cost model that acts as proxy for the actual execution time. Since the costs are directly reflected in the query execution time, optimal or near-optimal join orders are indispensable for the overall performance. However, the problem is inherently NP-hard [18]. This means that, unless $P = NP$, one has to resort to the exponential (exact) algorithm for small queries and to greedy strategies otherwise.

Motivation & Research Question. In a seminal work, Selinger et al. [38] introduced the first dynamic program to (exactly) optimize the ordering problem. The key observation is that the optimal solution S^* for a set of relations P , called the problem, satisfies Bellman’s optimality principle [1], namely that S^* is computed from two disjoint subproblems P_1 and P_2 , with optimal solutions S_1^* and S_2^* , respectively. The naive algorithm, **DPsize**, runs in $O(4^n)$ -time, which can be reduced to $O(3^n)$ by a careful traversal of the subsets of a given set, algorithm known as **DPsub** [43, 42].

© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM. This is a minor revision of the paper “DPconv: Super-Polynomially Faster Join Ordering” published in Proc. ACM Manag. Data, Vol. 2, No. 6 (SIGMOD), Article 234 (December 2024). DOI: <https://doi.org/10.1145/3698809>

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org. Copyright 2025 ACM 0001-0782/24/0X00 ...\$5.00.

$$\text{DP}[\overline{R_1 R_2 R_3 R_4}] = \min \left\{ \begin{array}{l} \text{DP}[\overline{R_1}] + \text{DP}[\overline{R_2 R_3 R_4}] \\ \text{DP}[\overline{R_2}] + \text{DP}[\overline{R_1 R_3 R_4}] \\ \text{DP}[\overline{R_3}] + \text{DP}[\overline{R_1 R_2 R_4}] \\ \text{DP}[\overline{R_4}] + \text{DP}[\overline{R_1 R_2 R_3}] \\ \text{DP}[\overline{R_1 R_2}] + \text{DP}[\overline{R_3 R_4}] \\ \text{DP}[\overline{R_1 R_3}] + \text{DP}[\overline{R_2 R_4}] \\ \text{DP}[\overline{R_1 R_4}] + \text{DP}[\overline{R_2 R_3}] \end{array} \right.$$

$$\stackrel{V := \overline{R_1 R_2 R_3 R_4}}{=} \min_{\emptyset \subset S \subset V} (\text{DP}[S] + \text{DP}[V \setminus S])$$

Subset Convolution

Figure 1: How join ordering dynamic programming algorithms, e.g., **DPsub**, are implicitly using subset convolution. However, they are computing it *naively*. **DPconv** instead uses a highly tuned implementation of *fast subset convolution*.

Later, Moerkotte and Neumann [26] showed that one can obtain an improved algorithm if one disallows cross-products, namely by considering the connectivity structure of the underlying query graph (the algorithm was later extended to query hypergraphs [27]). Their algorithm, **DPccp**, achieves the lower bound on the number of connected complement pairs which any dynamic program needs to traverse, as shown by Ono and Lohman [31]. Recently, Haffner and Dittrich [15] proved that join ordering reduces to computing shortest paths in an exponential-size graph in which the vertices are relation subsets. Their reduction enables the use of well-known speedups via heuristic search, as known from the A^* algorithm [16]. However, while their average-case running time beats that of **DPccp**, the worst-case running time still remains $O(3^n)$. The $O(3^n)$ -time bottleneck leads us to our main question:

Is there a way to break the seemingly unyielding $O(3^n)$ -time barrier?

Surprisingly, there is. To this end, consider Fig. 1, in which we show how the standard join ordering dynamic programming algorithms **DPsub** [43, 42] and **DPccp** [26] optimize the full set of relations $V = \{R_1, R_2, R_3, R_4\}$. Simply put, the algorithm iterates over all possible ways to split the original set V into two subsets. *This is exactly a subset convolution*. However, all current join ordering algorithms, **DPsize**, **DPsub**, and **DPccp**, perform it naively, i.e., the expression is evaluated as is. Fortunately for our community, research in algorithm design has led to a *fast subset convolution* [2].

Intuitively, fast subset convolution no longer naively enumerates subsets, but instead uses an FFT-inspired strategy that avoids redundant computational steps in the naive evaluation.

In this work, we develop a new exact algorithmic framework based on fast subset convolution that has a super-polynomial speedup over **DPccp** / **DPsub**, breaking the long-standing $O(3^n)$ time-barrier for the first time. We instantiate the framework for two well-studied cost functions, C_{out} and C_{max} , which guarantee time- and space-optimality of query execution, respectively. Namely, the latter minimizes the sum of the intermediate join sizes, while the former minimizes the largest intermediate one. This results in an $O(2^n n^2 W n \log W n)$ -time algorithm for C_{out} , which is $O^*(2^n)$ when the largest join cardinality W is polynomial in n ,¹ and an $O(2^n n^3)$ -time algorithm for C_{max} ; the latter running time is independent of W . At a practical level, we show that the instantiation for C_{max} is up to 30x faster than the classic algorithm for clique queries of 17 or more relations.

The framework also allows to further reduce the optimization time for C_{out} , using an $(1+\varepsilon)$ -approximation algorithm, to $O^*(2^{3n/2}/\sqrt{\varepsilon})$ -time. Unlike our *exact* algorithm for the same cost function, the running time of this algorithm is independent of W , making it attractive for queries with huge join cardinalities.

Contributions. In this shortened article, we will focus on the following two key contributions:

1. We introduce a new exact algorithmic framework based on subset convolution which breaks the long-standing time-barrier of $O(3^n)$ for the first time.
2. We provide a practical instantiation of the framework for C_{max} , achieving an $O(2^n n^3)$ -time algorithm.

Organization. The rest of the paper is organized as follows: First, in §2, we formalize the problem of join ordering and that of fast subset convolution. Then, we describe the machinery behind fast subset convolution in §3, and, in §4, we introduce **DPconv** along with the novel connection between join ordering and subset convolution. As part of §5, we provide a simple and *practical* algorithm for C_{max} , which we evaluate in §6. We then outline related work in §7, provide a discussion in §8, and finally conclude in §9.

2. BACKGROUND

In this section, we formalize both problems, namely join ordering and subset convolution.

2.1 Query Graph

Let $\mathcal{D} = \{R_1, \dots, R_n\}$ be a database that contains n relations. A select-project-join query $\mathcal{Q}$ is defined as

$$\mathcal{Q} = \Pi_A(\sigma_P(R_1 \times \dots \times R_n)), \quad (1)$$

where P is the conjunction of predicates that can be both join predicates, i.e., $R_i.a = R_j.b$, and selection predicates, i.e., $R_i.a = \text{const}$, and A is the list of attributes required to appear in the output. The operators Π , σ , and $\times$ are the projection, selection and cross-product operators, respectively, as defined in the relational algebra [6].

¹The notation O^* hides polynomial factors in the query size.

We can model a query as a *query graph* $Q = (V, E)$, where the vertex set V corresponds to the set of relations $\{R_i\}_{i \in [n]}$ of the query and the edge set $E = \{\{R_u, R_v\} \mid R_u, R_v \in V\}$ corresponds to the join predicates (called join edges in the sequel). Intuitively, a query can be evaluated by repeatedly joining two relations and replacing one of them with their join. Another prominent way of executing joins by worst-case optimal joins, which are not necessarily *binary* joins anymore [30]. In this work, we only concentrate on query optimization of binary joins. In this case, the order in which the joins are performed can be represented by a (binary) *join tree*, where the leaf nodes are the relations and the inner nodes are the corresponding joins.

2.2 Cost Function

To optimize the join order, one introduces a cost function $\mathcal{C}$ which best models the query execution time. The goal is to minimize the cost function among all possible join trees. Due to the binary structure of a join tree, the cost function can be computed recursively along a join tree $\mathcal{T}$, as follows:

$$\mathcal{C}(\mathcal{T}) = \begin{cases} 0, & \text{if } \mathcal{T} \text{ is a single relation} \\ c(T) \otimes \mathcal{C}(\mathcal{T}_1) \otimes \mathcal{C}(\mathcal{T}_2), & \text{if } \mathcal{T} = \mathcal{T}_1 \bowtie \mathcal{T}_2, \end{cases} \quad (2)$$

where c is the join cardinality function defined on sets of relations, $\otimes$ is an operator that combines the cost-values, T is the set of relations spanned by the join tree $\mathcal{T}$,² and $\mathcal{T}_1$ and $\mathcal{T}_2$ are the left and right join subtrees of $\mathcal{T}$, respectively.

Let us instantiate Eq. (2) for two cost functions, C_{out} and C_{max} , which guarantee time- and space-optimality of the query execution, respectively:

$$C_{\text{out}}(T) = c(T) + C_{\text{out}}(\mathcal{T}_1) + C_{\text{out}}(\mathcal{T}_2), \quad (3)$$

$$C_{\text{max}}(T) = \max\{c(T), C_{\text{max}}(\mathcal{T}_1), C_{\text{max}}(\mathcal{T}_2)\}. \quad (4)$$

We can observe that the “ $\otimes$ ” operator from above has been substituted by “+” and “max”, respectively.

2.3 Join Ordering and Dynamic Programming

By Bellman’s optimality principle [1], the problem of finding the optimal join tree $\mathcal{T}^*$ amounts to finding the optimal *split* of a set of relations S into two disjoint sets S_1 and S_2 , i.e., $S_1 \cap S_2 = \emptyset$ and $S = S_1 \cup S_2$. Consequently, given a cost function $\mathcal{C}$, the problem can be optimized by the following dynamic programming (DP) recursion, which closely follows the definition of $\mathcal{C}$:

$$\text{DP}(S) = \begin{cases} 0, & \text{if } |S| = 1 \\ c(S) \otimes \min_{\emptyset \subset T \subset S} (\text{DP}(T) \otimes \text{DP}(S \setminus T)), & \text{otherwise.} \end{cases} \quad (5)$$

Indeed, this is the idea explored by Selinger and the subsequent work [38, 43, 42, 26]. In particular, **DPccp** [26] implements the recursion by considering only sets of relations that induce a connected subgraph; for clique queries, **DPsub** and **DPccp** are both exactly the recursion above. As motivated in Fig. 1, Eq. (5) is a subset convolution. All previous algorithms evaluate it in the naive way, which takes $O(3^n)$ -time. **DPconv** speeds up its computation by employing fast subset convolution [2].

²Having a separate notation for the join tree and the set of relations it spans will prove useful in the following sections.

2.4 Subset Convolution

Subset convolution is one of the important tools in the field of exact algorithms [14, 7]. Its fast counterpart, called Fast Subset Convolution (FSC) [2], represented a breakthrough in the field by reducing the running time from the straightforward $O(3^n)$ to a non-trivial $O(2^n n^2)$.

Dynamic Programming Speedups. The main application of FSC is the speedup of several dynamic programming recursions of well-known NP-hard problems, such as the Steiner tree problem [9] and min-cost k -coloring [7]. While these problems may seem foreign to our research area, there is a striking similarity between the dynamic programming recursion of these problems and that of the join ordering problem. Indeed, they all implicitly use a subset convolution. Through our work, join ordering is now becoming part of this family of problems [37, 3, 7, 9, 32].

While our main result mostly uses FSC in a black-box manner, we present the full machinery behind it in §3. Note that for an efficient implementation of the dynamic programs, we refer the interested reader to Ref. [40, Sec. 5], as part of which we shaved a linear factor from the running time of generic FSC-based dynamic programs, as well as several constant factors hidden behind the running time's big-O notation; in the sequel, we refer to it as LAYEREDDP.

Key Idea. Let us first gain an intuition about how subset convolution works at a high level. First, note that the DP-table is by definition a set function: It maps subsets of relations to their corresponding costs. The usual way to refer to a subset structure is by a subset lattice, in our case of order n , since we have n relations. This leads to the following setting: Let f and g be two set functions on the subset lattice of order n , their subset convolution in the $(+, \times)$ ring is defined for all $S \subseteq [n] := \{1, \dots, n\}$ by

$$h(S) = (f * g)(S) = \sum_{T \subseteq S} f(T)g(S \setminus T).$$

Next we describe the low-level workings of FSC.

3. FAST SUBSET CONVOLUTION

We take a closer look at FSC from a practical perspective. In the following, we adopt the notation from the Parameterized Algorithms book [7], since it has established itself in the literature compared to that of Björklund et al. [2].

3.1 Zeta Transform

A fundamental operation in FSC is the zeta transform, defined as

$$(\zeta f)(S) = \sum_{T \subseteq S} f(T), \quad (6)$$

for any $S \subseteq [n]$. That is, the zeta transform sums f at all subsets of S . Naively, this can be computed in $O(3^n)$ -time for all $S \subseteq [n]$. However, we can compute it in $O(2^n n)$ -time by observing that we can reuse the computation done for subsets. We detail this in §3.4.

3.2 Ranked Convolution

Given ζf and ζg , the zeta transform of the actual convolution $h = f * g$, i.e., ζh , can now be computed point-wise. To this end, Björklund et al. [2] employ a *ranked* convolution.

Formally,

$$(\zeta h)(S, r) = \sum_{d=0}^r (\zeta f)(S, d)(\zeta g)(S, r-d), \quad (7)$$

for any $S \subseteq [n]$, where $|S| = r$. Thus, we have to apply a zeta transform for *each* rank, i.e., for each cardinality in $\{0, \dots, n\}$. The ranked convolution can then be computed naively in $O(2^n n^2)$, as for each rank r we need to iterate over all $d \leq r$.

3.3 Möbius Transform

To obtain the actual convolution, one applies the Möbius transform rank-wise. The Möbius transform is indeed the *inverse* of the zeta transform, i.e., $\zeta \mu = \mu \zeta = \text{id}$, and is defined for any $S \subseteq [n]$ as

$$(\mu f)(S) = \sum_{T \subseteq S} (-1)^{|T|} f(T). \quad (8)$$

A full-fledged example of FSC is shown in §3.5 and its associated Fig. 2.

3.4 Implementation

Zeta Transform. A naive evaluation of Eq. (6) leads to an $O(3^n)$ -time algorithm, as for each subset we are to sum up along all its subsets. However, there is a faster way computing it, commonly referred to as Yates' algorithm [46]. Define $\hat{f}_0(S) = f(S)$ for all $S \subseteq [n]$, and then iterate for all $j = 1, 2, \dots, n$ and $S \subseteq [n]$ as follows [2]:

$$\hat{f}_j(S) = \begin{cases} \hat{f}_{j-1}(S) & \text{if } j \notin S, \\ \hat{f}_{j-1}(S \setminus \{j\}) + \hat{f}_{j-1}(S) & \text{if } j \in S. \end{cases} \quad (9)$$

By induction, one can show that $\hat{f}_n(S) = (\zeta f)(S)$ for all $S \subseteq [n]$. The computation of Eq. (9) takes $O(2^n n)$ operations, as for each subset S we need to iterate over its elements. Lst. 1 shows an implementation of Eq. (9).³

```

1 zeta(f):
2   for (d = 0; d != n; ++d):
3     for (S = 0; S != 2**n; ++S):
4       if S & 2**d:
5         f[S] += f[S ^ 2**d]
```

Listing 1: Zeta transform

Möbius Transform. The Möbius transform can be computed in a similar way. Define $\check{f}_0(S) = f(S)$ for all $S \subseteq [n]$, and then evaluate the following recursion [2]:

$$\check{f}_j(S) = \begin{cases} \check{f}_{j-1}(S) & \text{if } j \notin S, \\ -\check{f}_{j-1}(S \setminus \{j\}) + \check{f}_{j-1}(S) & \text{if } j \in S. \end{cases} \quad (10)$$

Then one can show that $\check{f}_n(S) = (\mu f)(S)$ for all $S \subseteq [n]$ and the computation happens in $O(2^n n)$ operations as well.

A sketch of the entire FSC algorithm is shown in Lst. 3. We use the already-established Python's slicing notation “:” to denote an entire axis of the array, in our case, indexed by bitsets corresponding to the actual sets of relations. We next show a working example.

³G. Moerkotte made us aware that the zeta transform can, intriguingly, be implemented slightly faster using the complete “design matrix” [17], i.e., the binary matrix that encodes the subset relation.

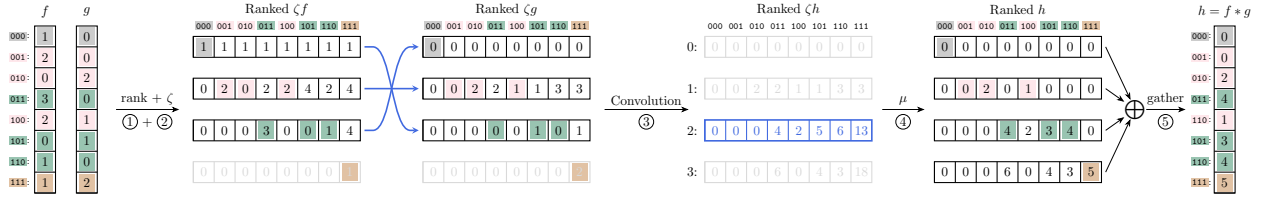


Figure 2: Visualizing the fast subset convolution (FSC), outlined in Lst. 3: We ① rank the set functions f and g and ② apply the zeta transform to obtain ζf and ζg , respectively. Then, we ③ perform the ranked convolution between ζf and ζg and ④ apply the Möbius transform to obtain the ranked h . Finally, we ⑤ reconstitute $h = f * g$, the actual subset convolution. We highlight in (this) color the steps needed to compute the rank “slice” $(\zeta h)(:, 2)$ of ζh during ranked convolution (as in §3.2). Intuitively, we need to sum up the dot products between the corresponding slices, i.e., $(\zeta f)(:, 0)$ with $(\zeta g)(:, 2)$, $(\zeta f)(:, 1)$ with $(\zeta g)(:, 1)$, and $(\zeta f)(:, 2)$ with $(\zeta g)(:, 0)$.

```

1 FSC(f, g):
2   for (r in [0, n]):
3     // Rank f and g
4     for (S with |S| = r):
5       f[S, r] = f[S]
6       g[S, r] = g[S]
7
8   // Zeta transform
9   zf[S, r] = zeta(f[:, r])
10  zg[S, r] = zeta(g[:, r])
11
12  // Ranked convolution
13  for (d in [0, r]):
14    zh[:, r] += zf[:, d] * zg[:, r - d]
15
16  // Moebius transform
17  h[:, r] = mu(zh[:, r])
18
19  // Reconstitute h
20  for (S in [0, 2**n]):
21    h[S] = h[S, |S|]
```

Figure 3: Fast subset convolution, visualized in Fig. 2.

Running Time. Let us calculate the total running time of FSC: The n zeta and Möbius transforms take in total $O(2^n n^2)$ -time, while the rank convolution itself takes $O(2^n n^2)$ -time.

3.5 Example

To facilitate the understanding of how fast subset convolution works, we provide a working example in Fig. 2. In particular, we visualize the steps of Lst. 3. Our example considers two set functions, f and g , of size 8, i.e., a subset lattice of size 3. Note that we have combined steps ① and ② in Fig. 2 into a single step.

① **Rank.** In the first step, we create as many rank “slices” as there are set cardinalities; in our case, there are 4 rank slices in total. Initially, they all contain only the values corresponding to the positions of the same cardinality. For instance, the slice corresponding to rank 2 is initially comprised of the values at positions 011, 101, and 110. Accordingly, these positions and values are displayed in the same color.

② **Applying Zeta.** Once we created the rank slices, we can now apply the zeta transform to them. Recall its definition in Eq. (6): For each set S , we sum all the values of f (and analogously for g) of at the indices of S ’s subsets. To show

this, consider the same rank slice 2 of ζf : The value at 111 – which is 4 – is the sum of 3 + 1. Similar in the rank slice 1 of ζg : The value at $(\zeta g)(111, 1)$ is made up of the non-zero values $g(010)$ and $g(100)$.

③ **Ranked Convolution.** Once both ranked ζf and ζg have been computed, we can run the (ranked) convolution between them, as described in Eq. (7). We visualize the steps for computing the rank slice 2 of ζh in Fig. 2, namely: The colored arrows connecting $(\zeta f)(:, 0)$ with $(\zeta g)(:, 2)$, $(\zeta f)(:, 1)$ with $(\zeta g)(:, 1)$, and $(\zeta f)(:, 2)$ with $(\zeta g)(:, 0)$ show that we need to multiply these rank slices to obtain $(\zeta h)(:, 2)$. As pointed out in Eq. (7), we simply perform a dot product between these and sum up the results. For instance, to obtain $(\zeta h)(111, 2)$, we need to perform the following calculation: $1 \cdot 1 + 4 \cdot 3 + 4 \cdot 0 = 13$.

④ **Applying Möbius.** To obtain the actual “ranked” h , we have to apply the Möbius transform onto ranked ζh . As explained in §3.3, the Möbius transform is the *inverse* of the zeta transform. Once this is done, the next paragraph explains how to obtain the final subset convolution result, h . The Möbius transform is applied as in Eq. (8), namely we consider all the subsets of a set S and *subtract* the values where the subset cardinality is odd, and *add* those for even cardinality. For instance, $h(111, 2)$ is computed as follows: The values $\zeta h(100, 2)$, $\zeta h(111, 2)$ are at odd cardinalities, so we subtract their values, while $\zeta h(011, 2)$, $\zeta h(101, 2)$, and $\zeta h(110, 2)$ are at even cardinalities, so we add them. In total, this results in $-2 - 13 + 4 + 5 + 6 = 0$, which is exactly $h(111, 2)$.⁴

⑤ **Gather.** Finally, once the ranked h has been fully computed by applying the Möbius transform to ζh , we can obtain the final h by taking a reverse process to step ①: Instead of scattering the set functions to rank slices, we now gather the rank slices into one set function. This is done by simply taking the positions from the corresponding rank slice and putting these into h ; this is also highlighted by the corresponding colors. For instance, to collect the positions 011, 101, and 110, we take them from the rank slice 2, since all these subsets have cardinality 2.

This concludes our example. In the following, we relate the join ordering problem to fast subset convolution for the first time, and provide a unified framework that can be instantiated for several cost functions.

⁴We corrected the calculation after receiving an anonymous feedback.

Algorithm 1: DPconv: Using fast subset convolution (FSC) to gradually optimize the dynamic programming table.

```

1: Input: Query graph  $Q = (V, E)$ , cardinality function  $c$ 
2: Output: Optimal cost value w.r.t.  $\mathcal{C}$ 
3:  $DP[\emptyset] \leftarrow +\infty$ 
4:  $DP[\{R_i\}] \leftarrow 0, \forall R_i \in V$ 
5: for each  $k$  in  $2, \dots, |V|$  do
6:    $DP' \leftarrow FSC_{(\min, \otimes)}(DP, DP)$ 
7:    $DP[S] \leftarrow DP'[S] \otimes c(S), \forall S \text{ s.t. } |S| = k$ 
8: end for
9: return  $DP[V]$ 

```

4. OUR FRAMEWORK

Let us consider the optimization of an *arbitrary* cost function $\mathcal{C}$ in its associated $(\min, \otimes)$ semi-ring under a generic framework.⁵ We will then instantiate the framework for C_{out} and C_{max} , respectively.

4.1 Join Ordering Meets Subset Convolution

The key observation behind our results is the (now trivial) observation that the definition of DP-recursion, Eq. (5), is similar to that of subset convolution. In particular, we show that join ordering falls into the category of dynamic programs which fast subset convolution has already been applied to. In our specific context, there are a few (minor) issues that need to be addressed for FSC to be applicable, issues that have also been considered by Björklund et al. [2] for other problems, namely:

- (i) The dynamic program DP, Eq. (5), is defined recursively.
- (ii) The subset T of S in the same Eq. (5) must not take $\emptyset$ nor S as value.

Overview. Both issues are resolved by a simple technique: We apply FSC layer-wise, i.e., we optimize sets of size 2 first, then those of size 3, and so on – this is what we call a *layer*. Specifically, at each layer k , since the DP-table has been computed for layers $k' < k$, we can directly optimize $DP[S]$ for all S with $|S| = k$ by a call to FSC. To alleviate issue (ii), we set $DP[\emptyset]$, i.e., the DP-cell representing the empty set of relations, to $+\infty$. Since FSC is called n times, the total optimization time adds up to $O(2^n n^3 \tau_C)$, where the function τ_C is tailored to the specific cost function $\mathcal{C}$ and accounts for the time overhead of semi-ring operations. We will cover the exact expressions of τ_C for individual cost functions in the following sections (see §4.3 for C_{out} and §4.4 for C_{max}).

Note that our improved LAYEREDDP [40, Sec. 5] shaves an $O(n)$ -factor off the running time, resulting in a total running time of $O(2^n n^2 \tau_C)$ for a cost function $\mathcal{C}$. We skip the details and refer the interested reader to the full paper.

Pseudocode. In Alg. 1, we outline the pseudocode behind our framework DPconv. It takes the query graph Q and the join cardinality function c as input and outputs the optimal cost value w.r.t. the specific cost function $\mathcal{C}$ which the $(\min, \otimes)$ semi-ring corresponds to.

The algorithm first optimizes the base cases, namely for the empty set of relations and for all sets containing only

⁵We ask the reader to not worry about the semi-ring notation.

Algorithm 2: BuildJoinTree: Recursively extracting the optimal bushy join tree from the DP-table

```

1: Input: Subset of relations  $S$ , DP-table
2: Output: The optimal bushy join tree
3: if  $|S| = 1$  return  $S$  end if
4: for each  $\emptyset \subset T \subset S$  do
5:   if  $c(S) \otimes DP(T) \otimes DP(S \setminus T) = DP(S)$  then
6:     return (BuildJoinTree( $T$ ), BuildJoinTree( $S \setminus T$ ))
7:   end if
8: end for

```

one relation. The former are initialized with $+\infty$, as argued above, and the latter with 0, cf. Eq. (5). Then, at each layer k , we optimize the subsets of size k by calling FSC on the current state of the DP-table (line 6) and then update the values with the join cardinalities of the subsets (line 7). To this end, note that Alg. 1 can optimize for cross-products out of the box: We simply need to also use the cardinalities of all cross-products in c . The running time remains naturally the same. Finally, we return the optimal cost represented by $DP[V]$.

Join Tree Extraction. Note that unlike previous algorithms, Alg. 1 does not maintain an OPT-table that stores the optimal split for each subset S . This is because FSC itself does not keep track of this information during its execution. In contrast, after the DP-table is fully-optimized, we can extract the optimal join tree from the DP-table itself, as outlined in Alg. 2. Specifically, for each set S , we find the subset T that was *intrinsically* used in FSC to optimize S , i.e., $DP[S] = DP[T] + DP[S \setminus T]$. Since there are at most n levels of recursion, the worst-case running time for Alg. 2 reads $O(2^n n)$.

Cross-Products. While allowing cross-products can lead to better overall costs [31, 33], the search space increases exponentially [31]. A prominent way to deal with cross-products is to heuristically insert them when they are guaranteed to be beneficial [31]; this tends to be the case when the estimated cardinality of the input is small enough [23, 33].

A natural question is whether DPconv could also support the optimization of cross-products, and whether this particular optimization would take more time than previously specified. Similar to DPsub [43, 42], we can use the cardinalities of the cross-products directly in c (line 7, Alg. 1). This means that DPconv can optimize for cross-products for both cost functions without any overhead.

We now come to the embedding technique, motivated and mentioned in §2.4, that helps us leverage the running time of the fast subset convolution to our employed semi-rings.

4.2 Embedding Technique

The embedding technique maps the values of the set functions to monomials and then runs the fast subset convolution algorithm in the $(+, \times)$ ring. The convolution values can then be read from the resulting polynomials. To see why this works, consider the functions [2, 1, 3, 4] and [5, 0, 1, 2]. When we embed these values to monomials, we obtain $[x^2, x^1, x^3, x^4]$ and $[x^5, x^0, x^1, x^2]$, respectively. Thus, by running their subset convolution $[x^2, x^1, x^3, x^4] * [x^5, x^0, x^1, x^2]$, we can retrieve the final values as follows: Consider the value at 001, which is $x^{2+0} + x^{1+5}$. Note that multiplication between monomials is simply an addition at

the exponent level, while the minimum—in our case, $\min\{2+0, 1+5\}$ —is represented by the smallest exponent in the resulting polynomial.

Representation. To allow for a seamless instantiation of our framework for other cost functions, we represent the polynomials in *coefficient form*, i.e., pairs of exponents and their associated coefficients. For instance, we represent $2x + 3x^4$ as $\{(1, 2), (4, 3)\}$.

Limitation. The core limitation of the embedding technique is that the size of the coefficient forms exactly corresponds to the largest input value. The reason is that value will be the largest exponent in the entire embedding of the corresponding set function.

In the following, we instantiate the framework for C_{out} and C_{max} , respectively. In §5, we show a simpler algorithm to optimize for C_{max} that bypasses the need for the embedding technique.

4.3 Instantiating C_{out}

In the case of C_{out} , we are working in the $(\min, +)$ semi-ring. To implement the embedding, we simply need to specify how the “+” operator should work—in the most general form, the “ $\otimes$ ” operator; compare Eq. (2). This corresponds to polynomial multiplication in the coefficient form. Let P_1 and P_2 be two polynomials in coefficient form. Then $P_1 \otimes P_2$ for an exponent e is defined as

$$(P_1 \otimes P_2)(e) = \sum_{\substack{(e_1, c_1) \in P_1 \\ (e_2, c_2) \in P_2 \\ e_1 + e_2 = e}} c_1 c_2. \quad (11)$$

Since the maximum value of the C_{out} cost function could be Wn (recall that W is the largest join cardinality) and assuming a FFT-based implementation of the convolution in Eq. (11), the factor τ_{out} for supporting C_{out} is $O(Wn \log Wn)$.

4.4 Instantiating C_{max}

We now specify the embedding for the $(\min, \max)$ semi-ring. Unlike C_{out} , we need to specify how the “max” operator should work. Namely, the coefficient of exponent e of two polynomials P_1 and P_2 in coefficient form reads:

$$(P_1 \otimes P_2)(e) = \sum_{\substack{(e_1, c_1) \in P_1 \\ (e_2, c_2) \in P_2 \\ \max(e_1, e_2) = e}} c_1 c_2. \quad (12)$$

The intuition is that all exponents *below* e contribute to its final coefficient. If used as in Eq. (12), the size of the coefficient form will still be W , as in the case of C_{out} ; this is prohibitively expensive. While there is indeed a way to mitigate this and obtain a running time of $O(2^n n^4)$, which is independent of W , we discovered a much simpler algorithm with an even better running time of $O(2^n n^3)$, which does not require the embedding technique. This is understandable due to the fact that, in the $(\min, \max)$ semi-ring, we are not creating *new* values, as is the case in C_{out} .

To not burden the reader with the technicalities of this approach, we will present directly the simpler algorithm as part of the next section. Apart from being simpler, the algorithm we will present is also practical, as we will show as part of §6.

Algorithm 3: Simpler DPconv[**max**]: Optimal cost w.r.t. C_{max} in $O(2^n n^3)$ -time

```

1: Input: Query graph  $Q = (V, E)$ 
2: Output: Optimal cost value w.r.t.  $C_{\text{max}}$ 
3:  $cs \leftarrow \text{sort}([c(S) \mid S \subseteq [V]], \text{decreasing}=\text{True})$ 
4:  $p, \text{step} \leftarrow 0, 2^{|V|-1}$ 
5: while  $\text{step} > 0$  do
6:    $\gamma \leftarrow cs[p + \text{step}]$ 
7:    $\text{DP} \leftarrow \text{LAYEREDDP}([c \leq \gamma])$ 
8:   if  $\text{DP}(V) > 0$  then
9:      $p \leftarrow p + \text{step}$ 
10:  end if
11:   $\text{step} \leftarrow \text{step} / 2$ 
12: end while
13: return  $cs[p]$ 

```

5. A SIMPLE ALGORITHM FOR C_{max}

While our framework can support C_{max} (see §4.4), we found a much simpler algorithm that does *not* require an intricate implementation.

Key Idea. The key insight is the following: Since we are applying only “min” and “max” operations, the optimal solution will take its value in the set of join cardinalities. Hence, we can *binary search* the optimal value OPT. To check whether a given value γ qualifies to be an optimal solution, we apply a technique used by Kosaraju for exact $(\min, \max)$ sequence convolution [20], which we will use on the DP-table itself. The strategy is to first put the DP-entries i.e.q. γ on 1 and those greater than γ on 0, and then run FSC, in the $(+, \times)$ ring, on this modified DP-table. In particular, this refers to one of the layers of the DP-table. We also use our improved layered dynamic programming, described in detail in Ref. [40, Sec. 5], which is a highly tuned implementation of the dynamic program for the $(+, \times)$ ring; for conciseness, we skip its technical details here.

Pseudocode. We outline the pseudocode of the algorithm in Alg. 3. It first *sorts* the join cardinalities in descending order and then performs a binary search on them, searching for the one which separates feasible γ ’s from infeasible ones (note that the maximum join cardinality is always feasible, but may not be the optimum). To this end, we employ Iverson’s bracket notation: Given a property P , $[P]$ returns 1 if the property is true, 0 otherwise. In our case, $[c \leq \gamma]$ is the following:

$$S \mapsto \begin{cases} 1, & \text{if } c(S) \leq \gamma, \\ 0, & \text{otherwise.} \end{cases}$$

Once the DP-table has been computed, the algorithm checks whether this value was feasible, i.e., whether V has a positive value in the DP-table. If that is the case, we search for smaller γ ’s. The algorithm concludes by returning the smallest γ for which $\text{DP}(V)$ is still positive. In the same manner as for the standard DPconv, we can build the join tree once we found the optimal value (see Alg. 2).

Running Time. Via LAYEREDDP [40, Sec. 5], our new Alg. 3 runs in time $O(2^n \log 2^n + 2^n n^2 \log 2^n) = O(2^n n^3)$. The additional factor $\log 2^n$ in the second term comes from the running time of the binary search on the 2^n -sized sorted list of join cardinalities.

6. EVALUATION

We show by means of experiments that **DPconv** achieves a significant speedup over the standard $O(3^n)$ -time join ordering algorithm.

Experimental Setup. We perform our experiments on a EC2 instance (c5.xlarge) which has an Intel Xeon Platinum 8275CL processor with 4 vCPUs and 8 GB of memory. All join ordering algorithms are implemented in C++.

Benchmark Sets. In the context of this shortened paper, we benchmark the algorithms on clique queries, for which we generate random join cardinalities $\leq 100M$, with the constraint that $c(S) \leq c(S_1)c(S_2), \forall S_1, S_2, \subseteq S, S_1 \cap S_2 = \emptyset, S_1 \cup S_2 = S$, i.e., we do not exceed the cardinality of the cross-product of any possible combination of subset pairs. Note that since we directly optimize on clique queries, the running times can also be considered as that of optimizing for cross-products, as discussed in §4.1. Moreover, since subset convolution does not (yet) exploit sparsity—in our case, corresponding to unconnected query subgraphs—the running time is independent of the cyclicity of the query graph (see §8 for a discussion on possible extensions).

Competitors. The standard exact algorithms, **DPccp** and **DPsub**, follow the implementation in the reproducibility experiment of Neumann and Radke [29].⁶ We also implement the bitsets as 64-bit integers, which we wrap with helper functions to provide iterators of subsets. The optimization time includes the time for extracting the join tree from the layered dynamic programming (see Alg. 2).

6.1 Super-Polynomial Speedup

Within our framework, **DPconv**, we have shown that join ordering can be done faster than $O(3^n)$. Specifically, we provided an $O(2^n n^2 W n \log W n)$ -time algorithm for optimizing C_{out} , which is $\tilde{O}(2^n)$ when the largest join cardinality W is polynomial in n , and an $O(2^n n^3)$ -time algorithm for optimizing C_{max} ; this is the first super-polynomial speedup for the join ordering problem. While the algorithm for C_{out} is not a practical one, we devised in §5 a simple and, *at the same time*, practical algorithm for C_{max} .

DPconv vs. DPsub. We benchmark on clique queries, as these are the hardest queries to optimize for [28]. In particular, **DPsub** excels at this type of queries since **DPccp** has the overhead of exploring the graph itself (note that this is also the case in the experiments of the original paper [26]). We show the optimization times for cliques of up to 24 relations in Fig. 4. The optimization time is averaged for each $n \in \{3, \dots, 24\}$ across 5 randomly generated instances.

The first observation is that our new algorithm is indeed practical: It starts being faster than **DPsub** after 17 relations, and for a large join query of 24 relations, it has a speedup of 29x. Note that $n = 17$ is still in the regime of the **JOB** benchmark. However, **JOB** has sparse query graphs, hence **DPccp** is enough for such queries.

7. RELATED WORK

The literature on join ordering is extensive. This is partly because of the effect that a bad join order can have on the query performance and hence the natural desire to avoid such cases. As a result, there are a few *exact* algorithms, a

⁶<https://db.in.tum.de/~radke/papers/hugejoins-reproducibility.pdf>

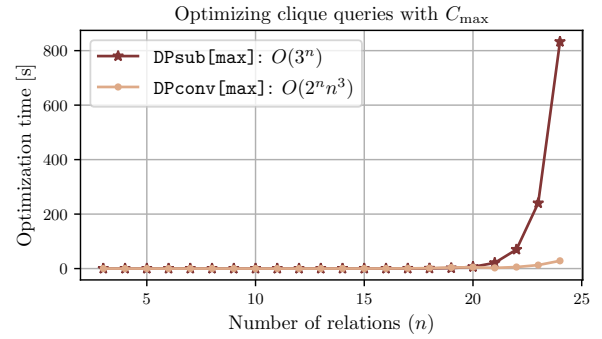


Figure 4: Clique queries optimization: Both **DPsub**[max] and **DPconv**[max] optimize for C_{max} .

small number of *polynomial-time* algorithms for restrictive cases, several *greedy* (non-optimal) algorithms, and a handful of optimizers based on general-purpose solvers. Our work falls into the category of exact algorithms. In particular, no previous work has observed the link to subset convolution, neither did it achieve a running time as we propose. We are the first to break the $O(3^n)$ time-barrier for the join ordering problem on generic query graphs (and bushy solutions). We divide the related work into exact, approximation, and best-effort algorithms. The latter are either polynomial-time algorithms for special instances or greedy algorithms without any approximation guarantee.

7.1 Exact Algorithms

The history of the join ordering problem starts at Selinger, proposing an $O(4^n)$ -time algorithm, commonly referred to as **DPsize** [38]. To some extent, this algorithm *does* subset convolution in the naive way, i.e., it iterates all subsets T of a given set S of relations, but does not do that in time $2^{|S|}$, but rather in time 2^n . Vance and Maier [43] observed this limitation and fixed it within the **DPsub** algorithm, which takes time $O(3^n)$. Since $O(3^n)$ seemed rather rigid, not being adaptive to the graph topology, Ono and Lohman [31] analyzed the *minimum* number of subplan pairs that have to be iterated in any dynamic program. To this end, Mørkotte and Neumann [26] designed **DPccp**, which emulates the graph topology and obtains as time-bound exactly the number of connected complement pairs ($\#ccp$'s). However, the running time $O(3^n)$ still persisted. In their recent work, Haffner and Dittrich [15] showed that using the A^* algorithm, one can obtain an algorithm which still outputs the optimal plan without having to explore all $\#ccp$'s. This is indeed a promising result, as it shows that the lower-bound of $\#ccp$ can in some cases be by-passed. However, in the worst case, the running time is still the unyielding $O(3^n)$. In our work, we obtain for the first time an $O^*(2^n W)$ -time algorithm, completely breaking the $O(3^n)$ time-barrier when W is polynomial in n . In the case of C_{max} , i.e., minimizing the maximum intermediate join cardinality, we obtain an $O(2^n n^3)$ -time algorithm, which is also practical.

Bottom-Up vs. Top-Down. It is well known that dynamic programs have two implementations, *bottom-up* and *top-down*, each with its advantages and disadvantages. One of the most compelling advantages of top-down enumeration

is the possibility of easily integrating cost-bounds so that the search space may be easily pruned [13]. Hence, Chaudhuri et al. [5] explore the possibility of implementing join ordering as a top-down procedure, only considering linear solutions. Building on this work, DeHaan and Tompa [8] extend the top-down method to bushy join trees, disallowing cross products. Fender and Moerkotte [11, 12] improve the running time of these algorithms and get rid of the connectedness check, i.e., only outputting the *ccp*'s.

7.2 Approximation Algorithms

Exact algorithms are rather expensive. To this end, Chatzergi et al. [4] analyzed whether there are instances that can be solved by approximation algorithms in polynomial time. Unless $P = NP$, the answer remains negative. Specifically, they showed that, for any $\delta > 0$, the problem of approximating the optimal cost K within a factor of $2^{\Theta(\log^{1-\delta} K)}$ is NP-hard. (Note that our $(1 + \varepsilon)$ -approximation algorithm in Ref. [40, Sec. 7] still runs in *exponential* time.)

7.3 Best-Effort Algorithms

The NP-hardness of a fundamental problem is a bitter truth. Hence, research has focused on finding polynomial-time algorithms for special instances or greedy algorithms for arbitrary query graphs.

Polynomial-Time Algorithms. Exponential-time algorithms fail to optimize larger queries in a reasonable time. To this end, it is interesting to ask which instances admit *polynomial-time* algorithms. The most notable one is the cubic-time algorithm for chain queries. Another class is that of tree queries, for which the IKKBZ algorithm returns the optimal left-deep join tree [18, 22]. Neumann and Radke [29] observed that one can use IKKBZ as a sub-routine: They *linearize* the query graph via IKKBZ (since a left-deep solution is inherently a linear ordering of the underlying graph) and then run the cubic-time dynamic program on top to build a near-optimal solution. This strategy yields excellent costs for tree queries.

Greedy Algorithms. Research has also focused on greedy algorithms which can at least *avoid* the bad plans. The most representative is the Greedy Operator Ordering (GOO) [10] that chooses the cheapest sub-plan at each step. This runs in $O(n \log n)$ -time, yet it does not come with any optimality guarantee on the output join order. This gap between exponential-time exact algorithms and purely greedy ones has remained unexplored until Kossman and Stocker [21] introduced Iterative Dynamic Programming (IDP) which refines the greedy join orders of large queries. The key insight is to iteratively run exact DP on join subtrees of size k .

General-Purpose Solvers. Join ordering has also been approached by several general-purpose solvers, such as genetic algorithms [39], mixed-integer linear programming [41], and simulated annealing [39]. Note that these works only approximate the optimal solution (*without* any approximation guarantee). The problem can also be optimized on quantum hardware via quantum annealing [45, 36]. However, this does not lower the *classical* time-complexity of exact join ordering. Motivated by the promise of workload-aware query optimization, research also has focused on *learned* alternatives: Marcus and Papaemmanouil [25] suggest using Reinforcement Learning and introduce an agent that outputs the join order and is penalized based on the corresponding

join cost. Motivated by the repetitiveness of the queries in cloud workloads [35], a further promising direction is query super-optimization [24].

8. DISCUSSION

Resource-Aware Query Optimization. The trend nowadays is to execute queries in multi-tenant cloud machines. Recently, Viswanathan et al. [44] made the case for *resource-aware* query optimization. The $C_{\max}$ cost function can serve as a proxy for the maximum memory consumption of a given query. Minimizing $C_{\max}$ of concurrently running queries can help reduce memory spikes.

Co-Optimizing C_{out} and $C_{\max}$. The cloud data warehouse Amazon Redshift uses predicted query memory to make scheduling decisions [34]. Instead, one could follow a proactive approach in which a query's runtime and memory consumption is co-optimized with query scheduling. For example, when there is a high (concurrent) memory load on the system, one would minimize the peak memory consumption of newly arriving queries, while when there is low memory load, one can afford a higher memory consumption. Likewise, if there are long-running queries with a low memory footprint in the system, one might opt for a query plan that uses more memory in exchange for faster execution.

More Practical Implementations. While we break the $O(3^n)$ time-barrier in the theoretical sense and, indeed, also provide a practical implementation for $C_{\max}$ running in $O(2^n n^3)$ -time, it is interesting to further explore practical implementations for C_{out} , both for the exact (§4.3) and the approximation algorithm [40, Sec. 7].

Sparse Subset Convolution. Subset convolution does not (yet) have a *sparse* counterpart, as is the case for sequence convolution (we refer the reader to Jin and Xu [19] for the latest results on sparse sequence convolution). This would be particularly useful for sparse query graphs.

9. CONCLUSION

Join ordering, or finding the optimal order of the joins of a query, is an indispensable task in a database management system. The problem has its roots in the seminal work of Selinger et al. [38], culminating with the graph-theoretic exact algorithm by Moerkotte and Neumann [26]. Despite recent research by Haffner and Dittrich [15], the worst-case running time has remained $O(3^n)$.

In this work, we showed the first super-polynomial speedup over the standard dynamic programming solution. Our framework optimizes (i) C_{out} in $O^*(2^n)$ -time, when the largest join cardinality W is polynomial in n , and (ii) $C_{\max}$ in $O(2^n n^3)$ -time. DPconv is based on subset convolution, a fundamental tool in parameterized algorithms [7], and uses the fact that join ordering is implicitly a dynamic programming recursion using subset convolution similar to other classic problems in the algorithm design literature (see Björklund et al. [2]).

Beyond the theoretical results, we have made DPconv practical for database systems. In particular, our algorithm for optimizing $C_{\max}$ outperforms the standard exact algorithm for cliques with 17 relations and more.

We expect future work on sparse subset convolution to further speed up our framework for query graphs with few connected subgraphs.

10. REFERENCES

- [1] R. E. Bellman. *The Theory of Dynamic Programming*. RAND Corporation, Santa Monica, CA, 1954.
- [2] A. Björklund, T. Husfeldt, P. Kaski, and M. Koivisto. Fourier meets möbius: fast subset convolution. In D. S. Johnson and U. Feige, editors, *Proceedings of the 39th Annual ACM Symposium on Theory of Computing, San Diego, California, USA, June 11-13, 2007*, pages 67–74. ACM, 2007.
- [3] A. Björklund, T. Husfeldt, and M. Koivisto. Set partitioning via inclusion-exclusion. *SIAM J. Comput.*, 39(2):546–563, 2009.
- [4] S. Chatterji, S. S. K. Evani, S. Ganguly, and M. D. Yemmanuru. On the complexity of approximate query optimization. In L. Popa, S. Abiteboul, and P. G. Kolaitis, editors, *Proceedings of the Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 3-5, Madison, Wisconsin, USA*, pages 282–292. ACM, 2002.
- [5] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim. Optimizing queries with materialized views. In P. S. Yu and A. L. P. Chen, editors, *Proceedings of the Eleventh International Conference on Data Engineering, March 6-10, 1995, Taipei, Taiwan*, pages 190–200. IEEE Computer Society, 1995.
- [6] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970.
- [7] M. Cygan, F. V. Fomin, L. Kowalik, D. Lokshtanov, D. Marx, M. Pilipczuk, M. Pilipczuk, and S. Saurabh. *Algebraic techniques: sieves, convolutions, and polynomials*, pages 321–355. Springer International Publishing, Cham, 2015.
- [8] D. DeHaan and F. W. Tompa. Optimal top-down join enumeration. In C. Y. Chan, B. C. Ooi, and A. Zhou, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007*, pages 785–796. ACM, 2007.
- [9] S. E. Dreyfus and R. A. Wagner. The steiner problem in graphs. *Networks*, 1(3):195–207, 1971.
- [10] L. Fegaras. A new heuristic for optimizing large queries. In G. Quirchmayr, E. Schweighofer, and T. J. M. Bench-Capon, editors, *Database and Expert Systems Applications, 9th International Conference, DEXA '98, Vienna, Austria, August 24-28, 1998, Proceedings*, volume 1460 of *Lecture Notes in Computer Science*, pages 726–735. Springer, 1998.
- [11] P. Fender and G. Moerkotte. A new, highly efficient, and easy to implement top-down join enumeration algorithm. In S. Abiteboul, K. Böhm, C. Koch, and K. Tan, editors, *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, pages 864–875. IEEE Computer Society, 2011.
- [12] P. Fender and G. Moerkotte. Reassessing top-down join enumeration. *IEEE Trans. Knowl. Data Eng.*, 24(10):1803–1818, 2012.
- [13] P. Fender, G. Moerkotte, T. Neumann, and V. Leis. Effective and robust pruning for top-down join enumeration algorithms. In A. Kementsietsidis and M. A. V. Salles, editors, *IEEE 28th International Conference on Data Engineering (ICDE 2012), Washington, DC, USA (Arlington, Virginia), 1-5 April, 2012*, pages 414–425. IEEE Computer Society, 2012.
- [14] F. V. Fomin and D. Kratsch. *Exact Exponential Algorithms*. Springer-Verlag, Berlin, Heidelberg, 1st edition, 2010.
- [15] I. Haffner and J. Dittrich. Efficiently computing join orders with heuristic search. *Proc. ACM Manag. Data*, 1(1):73:1–73:26, 2023.
- [16] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. Syst. Sci. Cybern.*, 4(2):100–107, 1968.
- [17] D. Havenstein, P. Lysakovski, N. May, G. Moerkotte, and G. Steidl. Fast Entropy Maximization for Selectivity Estimation of Conjunctive Predicates on CPUs and GPUs. In A. Bonifati, Y. Zhou, M. A. V. Salles, A. Böhm, D. Olteanu, G. H. L. Fletcher, A. Khan, and B. Yang, editors, *Proceedings of the 23rd International Conference on Extending Database Technology, EDBT 2020, Copenhagen, Denmark, March 30 - April 02, 2020*, pages 546–554. OpenProceedings.org, 2020.
- [18] T. Ibaraki and T. Kameda. On the optimal nesting order for computing n-relational joins. *ACM Trans. Database Syst.*, 9(3):482–502, 1984.
- [19] C. Jin and Y. Xu. Shaving logs via large sieve inequality: Faster algorithms for sparse convolution and more. In B. Mohar, I. Shinkar, and R. O'Donnell, editors, *Proceedings of the 56th Annual ACM Symposium on Theory of Computing, STOC 2024, Vancouver, BC, Canada, June 24-28, 2024*, pages 1573–1584. ACM, 2024.
- [20] S. Kosaraju. Efficient tree pattern matching. In *30th Annual Symposium on Foundations of Computer Science*, pages 178–183, 1989.
- [21] D. Kossmann and K. Stocker. Iterative dynamic programming: a new class of query optimization algorithms. *ACM Trans. Database Syst.*, 25(1):43–82, 2000.
- [22] R. Krishnamurthy, H. Boral, and C. Zaniolo. Optimization of nonrecursive queries. In W. W. Chu, G. Gardarin, S. Ohsuga, and Y. Kambayashi, editors, *VLDB'86 Twelfth International Conference on Very Large Data Bases, August 25-28, 1986, Kyoto, Japan, Proceedings*, pages 128–137. Morgan Kaufmann, 1986.
- [23] G. M. Lohman. Grammar-like functional rules for representing query optimization alternatives. In H. Boral and P. Larson, editors, *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 1-3, 1988*, pages 18–27. ACM Press, 1988.
- [24] R. Marcus. Learned query superoptimization. In R. Bordawekar, C. Cappiello, V. Efthymiou, L. Ehrlinger, V. Gadepally, S. Galhotra, S. Geisler, S. Groppe, L. Gruenwald, A. Y. Halevy, H. Harmouch, O. Hassanzadeh, I. F. Ilyas, E. Jiménez-Ruiz, S. Krishnan, T. Lahiri, G. Li, J. Lu, W. Mauerer, U. F. Minhas, F. Naumann, M. T. Özsu, E. K. Rezig, K. Srinivas, M. Stonebraker, S. R. Valluri, M. Vidal, H. Wang, J. Wang, Y. Wu, X. Xue, M. Zait, and K. Zeng, editors, *Joint Proceedings of Workshops at the 49th International Conference on Very Large Data*

- Bases (VLDB 2023)*, Vancouver, Canada, August 28 - September 1, 2023, volume 3462 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2023.
- [25] R. Marcus and O. Papaemmanouil. Deep reinforcement learning for join order enumeration. In R. Bordawekar and O. Shmueli, editors, *Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management, aiDM@SIGMOD 2018, Houston, TX, USA, June 10, 2018*, pages 3:1–3:4. ACM, 2018.
- [26] 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 U. Dayal, K. Whang, D. B. Lomet, G. Alonso, G. M. Lohman, M. L. Kersten, S. K. Cha, and Y. Kim, editors, *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006*, pages 930–941. ACM, 2006.
- [27] G. Moerkotte and T. Neumann. Dynamic programming strikes back. In J. T. Wang, editor, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, pages 539–552. ACM, 2008.
- [28] T. Neumann. Query simplification: graceful degradation for join-order optimization. In U. Çetintemel, S. B. Zdonik, D. Kossmann, and N. Tatbul, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*, pages 403–414. ACM, 2009.
- [29] T. Neumann and B. Radke. Adaptive optimization of very large join queries. In G. Das, C. M. Jermaine, and P. A. Bernstein, editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 677–692. ACM, 2018.
- [30] H. Q. Ngo, E. Porat, C. Ré, and A. Rudra. Worst-case optimal join algorithms: [extended abstract]. In M. Benedikt, M. Krötzsch, and M. Lenzerini, editors, *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pages 37–48. ACM, 2012.
- [31] K. Ono and G. M. Lohman. Measuring the complexity of join enumeration in query optimization. In D. McLeod, R. Sacks-Davis, and H. Schek, editors, *16th International Conference on Very Large Data Bases, August 13-16, 1990, Brisbane, Queensland, Australia, Proceedings*, pages 314–325. Morgan Kaufmann, 1990.
- [32] O. Ponta, F. Hüffner, and R. Niedermeier. Speeding up dynamic programming for some np-hard graph recoloring problems. In M. Agrawal, D. Du, Z. Duan, and A. Li, editors, *Theory and Applications of Models of Computation, 5th International Conference, TAMC 2008, Xi'an, China, April 25-29, 2008. Proceedings*, volume 4978 of *Lecture Notes in Computer Science*, pages 490–501. Springer, 2008.
- [33] B. Radke and T. Neumann. Lindp++: Generalizing linearized DP to crossproducts and non-inner joins. In T. Grust, F. Naumann, A. Böhm, W. Lehner, T. Härder, E. Rahm, A. Heuer, M. Klettke, and H. Meyer, editors, *Datenbanksysteme für Business, Technologie und Web (BTW 2019), 18. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme“ (DBIS), 4.-8. März 2019, Rostock, Germany, Proceedings*, volume P-289 of *LNI*, pages 57–76. Gesellschaft für Informatik, Bonn, 2019.
- [34] G. Saxena, M. Rahman, N. Chainani, C. Lin, G. Caragea, F. Chowdhury, R. Marcus, T. Kraska, I. Pandis, and B. M. Narayanaswamy. Auto-wlm: Machine learning enhanced workload management in amazon redshift. In S. Das, I. Pandis, K. S. Candan, and S. Amer-Yahia, editors, *Companion of the 2023 International Conference on Management of Data, SIGMOD/PODS 2023, Seattle, WA, USA, June 18-23, 2023*, pages 225–237. ACM, 2023.
- [35] T. Schmidt, A. Kipf, D. Horn, G. Saxena, and T. Kraska. Predicate caching: Query-driven secondary indexing for cloud data warehouses. In P. Barceló, N. Sánchez-Pi, A. Meliou, and S. Sudarshan, editors, *Companion of the 2024 International Conference on Management of Data, SIGMOD/PODS 2024, Santiago AA, Chile, June 9-15, 2024*, pages 347–359. ACM, 2024.
- [36] M. Schönberger, S. Scherzinger, and W. Mauerer. Ready to leap (by co-design)? join order optimisation on quantum hardware. *Proc. ACM Manag. Data*, 1(1):92:1–92:27, 2023.
- [37] J. Scott, T. Ideker, R. M. Karp, and R. Sharan. Efficient algorithms for detecting signaling pathways in protein interaction networks. In S. Miyano, J. P. Mesirov, S. Kasif, S. Istrail, P. A. Pevzner, and M. S. Waterman, editors, *Research in Computational Molecular Biology, 9th Annual International Conference, RECOMB 2005, Cambridge, MA, USA, May 14-18, 2005, Proceedings*, volume 3500 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2005.
- [38] 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 P. A. Bernstein, editor, *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data, Boston, Massachusetts, USA, May 30 - June 1, 1979*, pages 23–34. ACM, 1979.
- [39] M. Steinbrunn, G. Moerkotte, and A. Kemper. Heuristic and randomized optimization for the join ordering problem. *VLDB J.*, 6(3):191–208, 1997.
- [40] M. Stoian and A. Kipf. DPconv: Super-Polynomially Faster Join Ordering. *Proc. ACM Manag. Data*, 2(6):234:1–234:26, 2024.
- [41] I. Trummer and C. Koch. Solving the join ordering problem via mixed integer linear programming. In S. Salihoglu, W. Zhou, R. Chirkova, J. Yang, and D. Suciu, editors, *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 1025–1040. ACM, 2017.
- [42] B. Vance. *Join-order Optimization with Cartesian Products*. PhD thesis, Oregon Graduate Institute of Science and Technology, 1998.
- [43] B. Vance and D. Maier. Rapid bushy join-order

- optimization with cartesian products. In H. V. Jagadish and I. S. Mumick, editors, *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996*, pages 35–46. ACM Press, 1996.
- [44] L. Viswanathan, A. Jindal, and K. Karanasos. Query and resource optimization: Bridging the gap. In *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*, pages 1384–1387. IEEE Computer Society, 2018.
- [45] T. Winker, U. Çalikyilmaz, L. Gruenwald, and S. Groppe. Quantum machine learning for join order optimization using variational quantum circuits. In S. Groppe, L. Gruenwald, and C. Hsu, editors, *Proceedings of the International Workshop on Big Data in Emergent Distributed Environments, BiDEDE 2023, Seattle, WA, USA, 18 June 2023*, pages 5:1–5:7. ACM, 2023.
- [46] F. Yates. The design and analysis of factorial experiments. *Imperial Bureau of Soil Science*, 1937.

Automating Vectorized Distributed Graph Computation (Technical Perspective)

Julian Shun
MIT CSAIL
jshun@mit.edu

Graph data, which are used to model relationships between objects of interest, arise in many important applications today. For example, they are used to model human relationships in social networks, links among Web pages on the Internet, transactions in financial networks, interactions among proteins, genes, and molecules in biological/chemical networks, meshes in scientific simulations, particle interactions in physical/quantum systems, relationships among galaxies in astronomical studies, communication in cyber networks, and links in geospatial data. Analyzing the structure and properties of these graphs is an important component of many data analytics pipelines.

With the explosion in the volume of data, graphs have become very large and can contain hundreds of billions of vertices and trillions of edges. It is therefore crucial to design scalable solutions to enable graph analysis to be done efficiently. With the ubiquity of parallel machines today, using parallelism to speed up graph processing has become standard. Over the past several decades, parallel graph algorithms have been proposed for different classes of hardware platforms (CPUs, GPUs, distributed clusters, domain-specific accelerators, etc.). However, due to the complexity of parallel programming, developing efficient algorithms is time-consuming and is only accessible to a small subset of programmers who are experts in parallelism.

To make parallel graph processing accessible to non-experts, high-level programming frameworks have been proposed. These solutions provide high-level APIs that users can use to write parallel graph algorithms without having to deal with most of the complexities of parallel programming. Numerous graph processing frameworks have been proposed over the past two decades, targeting different types of hardware platforms, graph workloads, and graph data.

One kind of graph workload involves running multiple instances of a graph algorithm (e.g., with different source vertices). For example, one may be interested in computing shortest paths or local clusters from a collection of source vertices. A simple approach for tackling such workloads is to run an algorithm for each source vertex independently. However, it is often the case that the different algorithm instances involve some redundant computation, which can be avoided by interleaving the runs together. To this end,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright 2008 ACM 0001-0782/08/0X00 ...\$5.00.

several multi-instance graph algorithms have been proposed in the literature, and they show promising speedups over the simple baseline of running the instances independently. However, existing solutions require significant expertise to develop and understand, even when they are part of programming frameworks.

To address this challenge, the following paper [2] presents AutoMI, a framework that simplifies the task of developing distributed multi-instance graph algorithms. AutoMI automatically transforms vertex-centric graph algorithms written in the GAS (Gather-Apply-Scatter) programming model [1] into multi-instance algorithms with SIMD vectorization.

The paper first shows that a naive conversion of a single-instance graph algorithm into a multi-instance graph algorithm with SIMD vectorization can lead to programs that produce incorrect results. Then, it presents the AutoMI framework that ensures correct program generation. AutoMI creates an abstract syntax tree (AST) from the input program and then generates a vectorized multi-instance program according to the AST. The approach, by default, generates code for tracking the visited states of vertices. However, the authors observe that if a program satisfies a certain kind of idempotence, then the tracking information is not required for correctness. This leads to simpler and faster code. The notion of idempotence that is required for bypassing the tracking code is formalized via an algebraic characterization of GAS programs. The authors use AutoMI to generate multi-instance graph algorithms for integer breadth-first search (BFS), boolean BFS, single-source shortest paths, graph coloring, sparse matrix-vector multiplication, personalized PageRank, and collaborative filtering.

Multi-instance graph algorithms have many important applications, and the AutoMI framework is one of the latest additions to the collection of tools available for programming such algorithms. It would be interesting to see how this paper's ideas can be extended to other kinds of graph programs and hardware platforms.

1. REFERENCES

- [1] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, page 17–30, 2012.
- [2] W. Zhao, Y. Cao, P. Buneman, J. Li, and N. Ntarmos. Automating vectorized distributed graph computation. *Proc. ACM Manag. Data*, 2(6), Dec. 2024.

One Graph, Many Sources: Automating Vectorization Safely

Wenyue Zhao, Yang Cao, Peter Buneman

The University of Edinburgh, UK

{wenyue.zhao, yang.cao, peter.buneman}@ed.ac.uk

Jia Li, Nikos Ntarmos

Edinburgh Research Center, Huawei, UK

{jia.li3, nikos.ntarmos}@huawei.com

ABSTRACT

Many graph workloads repeatedly run the same traversal or iterative computation from many different source vertices. Multi-instance execution can share work across sources and, if implemented carefully, exploit SIMD by packing per-source state into vectors. However, in vertex-centric graph processing systems, naive “vectorize everything” transformations can be wrong: a vertex may be active for only a subset of sources in a round, yet unguarded SIMD updates implicitly advance all lanes, leading to incorrect answers.

We present **AutoMI**, a source-to-source compilation framework that automatically converts a single-instance program into a provably correct SIMD-vectorized multi-instance program runnable on existing vertex-centric engines. **AutoMI** uses a per-message bit-vector **track** to recover per-source activation and generate masked SIMD updates. It further provides a **TrackFree** optimization, guided by an algebraic idempotence characterization, that safely drops **track** and produces streamlined code when over-activation is harmless.

On six real graphs (up to billions of edges) and up to 256 sources, **AutoMI** achieves 9.6 to 29.5 $\times$ speedup over parallelized serial evaluation, 7.1 to 26.4 $\times$ over batch processing, and 2.6 to 4.6 $\times$ over prior handcrafted multi-instance code.

1. INTRODUCTION

Modern graph workloads rarely run a graph algorithm just once. In social analysis, web search and recommendation, it is common to execute the same traversal or iterative computation many times on the same graph but with different source vertices. Specifically, given a graph $G = (V, E)$ and a source-parametric query $Q(s)$, the task is to compute $Q(s_1), \dots, Q(s_k)$ for a set of sources $s_1, \dots, s_k$. Examples include classic traversal queries like reachability, breadth-first search (BFS), single-source shortest paths (SSSP), as well as advanced graph analytics like sparse matrix-vector

multiplication and graph learning [23], where computations are recast as iterative vertex updates on an induced graph.

Multi-instance algorithms. A straightforward approach is *serial evaluation* that runs a standard graph algorithm once per source, in parallel. While easy to implement, this repeats substantial work, both computation and communication (distributed setting), and fails to exploit a key observation: different sources often traverse overlapping regions of the graph. This has motivated *multi-instance* graph algorithms [33, 32, 12, 21, 29, 10, 30, 18, 35, 38, 22, 42], which interleave multiple instances to share traversal and computation. When instances execute similar operations on the same vertices, multi-instance execution opens door to SIMD vectorization acceleration by packing per-source state into vectors and applying the same instruction across instances.

Unfortunately, implementing multi-instance algorithms and obtaining these benefits can be notoriously challenging. Even for simple algorithms such as BFS, it takes substantial and delicate effort to ensure that the benefits of multi-instance computation outweigh the overhead of aligning the computations across sources. For instance, a multi-instance version of BFS requires approximately 600 lines of core code to be effective despite the simple algorithmic logic [35], without even applying SIMD vectorization yet [22]. A transformation that “packs per-source state into vectors and applies SIMD throughout” can be *incorrect*, even in vertex-centric frameworks [25, 23, 27] dedicated to parallel graph computations.

The core challenge is that, during graph traversal a vertex can be active for some sources but not others in the same round. Naive SIMD vectorization implicitly assumes that all lanes are active whenever a vertex function fires, which can corrupt per-source state (*e.g.*, marking a vertex as visited for sources that have not reached it yet) and lead to wrong answers. Preventing this would require extra work that carefully aligns the computations and progresses across sources, which, if not done carefully, may bring in overhead that outweigh the benefits of multi-instance implementation.

Automating multi-instance computations. In response to the challenge of developing effective multi-instance algorithms, we present **AutoMI**, a compilation framework that automatically converts a standard single-instance vertex-centric algorithm [2, 25, 17, 23, 15, 13] written in the GAS (Gather-Apply-Scatter) programming model [15, 13, 27, 36, 16, 24, 39, 11] into a *vectorized* multi-instance GAS program that runs directly on existing distributed engines. **AutoMI** ensures correctness by introducing a per-message bit-vector, **track**, which records *which sources* are logically responsible for a vertex activation. During code generation, **AutoMI** au-

© 2025. This is a minor revision of the paper “Automating Vectorized Distributed Graph Computation” published in Proc. ACM Manag. Data 2, 6 (SIGMOD), Article 254 (December 2024). DOI: <https://doi.org/10.1145/3698833>

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ALGORITHM 1: Single-instance Integer BFS (iBFS)

```

1 #define vertex_data { <int> ans, <bool> visited } ; // vertex
  properties
2 #define msg { <int> ans } ; // define message struct

3 Function Gather(vertex u, edge (u, v), vertex v) // invoked at v
  (center vertex)
4   return msg // sent from u by Scatter in previous round

5 Function Gather_Acc(vertex v, msg_acc, msg) // center vertex: v
6   msg_acc.ans ← Min (msg_acc.ans, msg.ans) ; // accumulate
  and aggregate msg in msg_acc; initially, msg_acc.ans is 0 if v is
  source and +∞ otherwise
7   return msg_acc;

8 Function Apply(vertex v, msg_acc) // center vertex: v
9   if NOT v.visited then
10    v.ans ← msg_acc.ans;
11    v.visited ← True;
12    changed ← True ; // mark changed as True

13 Function Scatter(vertex v, msg_acc) // center vertex: v
14   if changed then // check if changed is True
15    msg.ans ← v.ans + 1 ;
16    Signal (msg) ; // activate and send msg to outward neighbors

```

tomatically turns **track** into SIMD masks that guard vectorized updates, so each lane evolves exactly as it would under an independent single-source execution.

Over-vectorization and TrackFree. Unlike generic programs where incorrectly activated lanes are always harmful as they lead to incorrect answers, there are certain graph algorithms for which one can safely *over-vectorize*: we simply activate vertex functions for all sources even if not all of them have reached the vertex, and we assure that even if we “over-activate” vertices, the algorithm converges to the same fixed point as the correct multi-instance algorithm would do.

AutoMI captures this case via a **TrackFree** optimization that removes **track** entirely and generates streamlined, more efficient SIMD code. To decide when **TrackFree** is safe, we extend linear algebra to allow users to abstract algorithmic logic of the Apply and Scatter functions of their GAS programs in the extended algebra. We then develop a formal characterization of track-free vectorization of single-instance GAS algorithms by reducing to two *algebraic idempotence* properties of their algebra abstraction, allowing users to decide when to enable track-free vectorization in AutoMI.

On billion-edge real-life graphs, AutoMI-generated programs deliver substantial speedups (up to roughly $10\times$ – $30\times$ over serial evaluation, and up to $2.6\times$ – $4.6\times$ over handcrafted highly optimized multi-instance implementations)

Takeaway. Multi-instance algorithms can be highly efficient, but implementing them correctly and efficiently, especially when exploiting SIMD, is notoriously tricky. AutoMI makes “multi-instance + SIMD” practical without requiring developers to redesign algorithms or adopt specialized programming models. It does so via a source-to-source compilation framework that generates provably correct multi-instance GAS programs, plus a principled **TrackFree** optimization that identifies maximum performance benefits from SIMD vectorization without impairing correctness guarantees.

2. WHY NAIVE VECTORIZATION FAILS

Graph queries and computations. We consider a (typically large) directed graph $G = (V, E)$, possibly with edge properties such as weights. A query instance is parameterized by

ALGORITHM 2: Multi-instance Integer BFS

```

1 #define vertex_data { array <int> ans, bitvec <bool> visited } ;
2 #define msg { array <int> ans, bitvec <bool> track } ;

3 Function Gather(vertex u, edge (u, v), vertex v)
4   return msg ;

5 Function Gather_Acc(vertex v, msg_acc, msg)
6   msg_acc.track ← simd_Or (msg_acc.track, msg.track) ;
7   msg_acc.ans ← simd_mask_Min (msg.track, msg_acc.ans, msg.ans);
  return msg_acc;

8 Function Apply(vertex v, msg_acc)
9   mask ← simd_mask_Neg (msg_acc.track, v.visited) ;
10  v.ans ← simd_mask_Set (mask, v.ans, msg_acc.ans) ;
11  v.visited ← simd_mask_Set (mask, v.visited, 1) ; // 1: constant
  True vector
12  changed ← simd_mask_Set (mask, changed, 1) ;

13 Function Scatter(vertex v, msg_acc)
14  mask ← simd_mask_Set (msg_acc.track, mask, changed) ;
15  msg.ans ← simd_mask_Add (mask, v.ans, 1);
16  if NOT simd_All_Zero (mask) then
17    msg.track ← simd_Set (msg.track, mask) ;
18    Signal (msg) ;

```

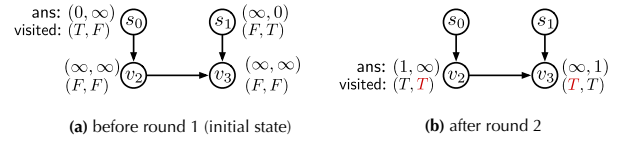


Figure 1: Data graph for Integer BFS (sources: s_0 and s_1)

a source vertex $s \in V$ and returns, for each vertex $v \in V$, an answer value determined by the query semantics. Classic traversal instances include reachability (Boolean BFS), levels (Integer BFS), and single-source shortest paths (SSSP), where information is propagated along edges from s .

Beyond traversals, AutoMI also targets common graph analytics kernels that can be expressed in the same vertex-centric style. We list some examples below. (a) Sparse Matrix-Vector multiplication (SpMV), which computes an output vector $y = Ax$ where A is (a weighted variant of) the adjacency matrix of G and x is an input vector with one entry per vertex (often source-dependent in personalized settings). Since A is $|V| \times |V|$, the result y is also a length- $|V|$ vector, *i.e.*, it returns one value $y[v]$ per vertex, obtained by aggregating contributions from neighbors, *e.g.*, $y[v] = \sum_{(u,v) \in E} w(u,v) x[u]$. (b) This is similar for Personalized PageRank (PPR), which iteratively applies SpMV-style propagation together with a source-specific personalization/restart distribution. (c) Collaborative filtering (CF), which iteratively updates per-vertex latent factors by stochastic gradient descent driven by edge interactions, can also be recast similarly. These problems are not always phrased as graph traversals, but they induce vertex-centric iterative computations and can be instantiated for many sources or personalization seeds in the same way as BFS and SSSP.

Vertex-centric programs. AutoMI targets vertex-centric graph systems (*e.g.*, Pregel [25], PowerGraph [15] and PowerLyra [13]) that run a user program “at each vertex” in synchronous rounds (*a.k.a.* supersteps) [25, 15, 13]. In the GAS (Gather-Apply-Scatter) model [15, 13, 26], an active vertex (i) aggregates messages from its neighbors (Gather), (ii) updates its local vertex properties (Apply), and (iii) decides whether to send messages and activate neighbors for the next round

Rule 1 Message Structure	
<code>#define msg { ... }</code>	$\Rightarrow$ <code>#define msg { ... <bool> track }</code>
Rules 2-5 Gather, Gather_Acc, Apply, Scatter Functions	
Function Name (args ...)	$\Rightarrow$ Function Name (args ...)
<code>stmt</code>	<code>mark(T) stmt</code>
Rule 6 Compound Statement	
<code>mark(ℓ) { stmt₁...stmt_n }</code>	<code>mark(ℓ) stmt₁...</code>
// ℓ : 'T' or new var (rule 7)	<code>mark(ℓ) stmt_n</code>
Rule 7 IF Selection Statement	
/* I_t : a distinct variable*/	
<code>mark(ℓ)</code>	<code>$I_t \leftarrow \ell \wedge E$</code>
<code>if E then stmt_t</code>	$\Rightarrow$ <code>if I_t then mark(I_t) stmt_t</code>
<code>else stmt_f</code>	<code>$I_f \leftarrow \ell \wedge \neg I_t$</code>
	<code>else mark(I_f) stmt_f</code>

Table 1: Annotation rules: each line on RHS of the rules corresponds to a node of the annotated AST.

for converting a given single-instance GAS program (e.g., Algorithm 1) into a provably correct multi-instance program (e.g., Algorithm 2) that directly runs on any GAS system.

Concretely, given a single-instance GAS program **A** and a set of k sources $s_1, \dots, s_k$, AutoMI produces a multi-instance program **A_m** that computes $Q(s_1), \dots, Q(s_k)$ in one run. The core challenge is to (a) deduce, at runtime, which sources are “visiting” a vertex v when it is activated, and (b) use this information to guard SIMD applications so that each lane evolves exactly as in an independent execution of **A**.

AutoMI introduces a new per-message bit-vector field **track**. Intuitively, **msg.track** records which sources are responsible for the activation carried by that message. As messages are aggregated in **Gather**, AutoMI maintains an accumulated mask **msg_acc.track** at each active vertex; **msg_acc.track** is then used as the SIMD mask that guards **Apply/Scatter** updates. This directly addresses the failure mode of the “populate-and-vectorize” vectorization in Section 2: even though the engine schedules at the vertex level, **track** reconstructs lane-level activation information needed for correctness.

AutoMI: single-instance $\Rightarrow$ multi-instance. AutoMI performs the source-to-source conversion in two steps.

- AST annotation: AutoMI first parses the input GAS program into an abstract syntax tree (AST) and annotates statements with labels that will later be interpreted as SIMD masks (Table 1).
- Vectorized code generation: AutoMI traverses the annotated AST and generates a vectorized multi-instance GAS program by (i) populating data structures and (ii) emitting SIMD operations guarded by masks derived from annotations (Table 2).

Step (1): AST annotation (Table 1). AutoMI first obtains an AST of the input GAS program via a compiler front-end, and then alters and annotates it via rules in Table 1:

(a) *Introduce track.* Rule 1 extends the message structure **msg** with a Boolean field **track**.

(b) *Annotate.* Rules 2-6 propagate a default label T through the statements of GAS functions. Intuitively, T means “execute under the current activation mask”; its exact interpretation depends on the phase, as we will see below.

(c) *Alter AST & refine labels.* Rule 7 rewrites an **if** statement under label ℓ into two per-branch masks $I_t \leftarrow \ell \wedge E$ and $I_f \leftarrow \ell \wedge \neg I_t$, and annotates then/else subtrees with I_t and

Rule 1 Structure Definition (using msg as example)	
Codegen (#define msg { ... <bool> track })	$\Rightarrow$ #define msg { ... bitvec<bool> track }
Rule 2 Gather_Acc Function Definition	
Codegen (Gather_Acc(args ...) mark(T) stmt)	$\Rightarrow$ Gather_Acc(args ...) msg_acc.track $\leftarrow$ simd_Or(msg_acc.track, msg.track) Codegen(mark(T) stmt)
Rule 3 Gather, Apply, Scatter Function Definition	
Codegen (Function Name (args ...) mark(T) stmt)	$\Rightarrow$ Function Name (args ...) Codegen(mark(T) stmt)
Rule 4 Signal Function Call	
Codegen (mark(ℓ) Signal(msg))	$\Rightarrow$ if $\ell \neq 0$ then /*NOT simd_All_Zero (ℓ)*/ msg.track $\leftarrow$ simd_Set(msg.track, ℓ) Signal(msg)
Rule 5 Assignment Statement	
Codegen (mark(ℓ) var $\leftarrow$ e)	$\Rightarrow$ var $\leftarrow$ simd_mask_Set(ℓ , var, e)
Codegen (mark(ℓ) var $\leftarrow$ Op(e_1))	$\Rightarrow$ var $\leftarrow$ simd_mask_Op(ℓ , e_1)
Codegen (mark(ℓ) var $\leftarrow$ e_1 Op e_2)	$\Rightarrow$ var $\leftarrow$ simd_mask_Op(ℓ , e_1 , e_2)
Rule 6 Compound Statement	
Codegen (mark(ℓ) stmt ₁ ... mark(ℓ) stmt _n)	$\Rightarrow$ { Codegen(mark(ℓ) stmt ₁) ... Codegen(mark(ℓ) stmt _n) }
Rule 7 IF Selection Statement	
Codegen ($I_t \leftarrow \ell \wedge E$ if I_t then mark(I_t) stmt _t $I_f \leftarrow \ell \wedge \neg I_t$ else mark(I_f) stmt _f)	$\Rightarrow$ Codegen($I_t \leftarrow \ell \wedge E$) Codegen(mark(I_t) stmt _t) Codegen($I_f \leftarrow \ell \wedge \neg I_t$) Codegen(mark(I_f) stmt _f)

Table 2: Code-generation rules: LHS denotes a node of the annotated AST and RHS is the generated code fragment.

I_f , respectively. This makes control flow lane-aware: each branch will execute only for lanes that satisfy its predicate.

Example 1: Figure 2 illustrates AutoMI’s two-step conversion on Algorithm 1. In the annotation step, AutoMI (i) extends **msg** with the **track** field (Rule 1) and (ii) propagates the default label T through the four GAS functions (Fig. 2a). In **Scatter** (Fig. 2b), it rewrites each **if** under label ℓ into per-lane masks $I_t \leftarrow \ell \wedge E$ and $I_f \leftarrow \ell \wedge \neg I_t$ (Rule 7). $\square$

Step (2): vectorized code generation (Table 2). Given the annotated AST, AutoMI generates a multi-instance GAS program by a depth-first traversal. Its code generator **Codegen** interprets annotations as SIMD masks via rules of Table 2.

(a) *Structure populating.* For k sources, each scalar vertex property (e.g., **ans**) is lifted to a length- k array storing per-source values; similarly for message fields. In addition, **msg** is extended with **track** now represented as a bit-vector (Rule 1).

(b) *Propagating track through GAS.* In **Gather_Acc**, AutoMI accumulates the incoming **msg.track** into **msg_acc.track** via bitwise OR in Rule 2. In **Scatter**, when a vertex signals neighbors, AutoMI sets **msg.track** to the current mask and only sends the message if at least one lane is active (Rule 4). These ensure that for each vertex v and round, **msg_acc.track** identifies exactly lanes that logically activate v .

(c) *Track-guarded SIMD updates.* Assignments and operations on vertex properties are compiled into SIMD operations guarded by the current mask label (Rules 5-7). It interprets T as an all-ones mask in **Gather/Gather_Acc** (since the phase is purely message transport/aggregation), and as **msg_acc.track** in **Apply/Scatter** (so updates occur only for active lanes). When Rule 7 introduces a branch mask vari-

ALGORITHM 3: Multi-instance Boolean BFS (bBFS)

```

1 #define vertex_data { bitvec <bool> ans };
2 #define msg { bitvec <bool> ans };

3 Function Gather(vertex u, edge (u, v), vertex v)
4   return msg;

5 Function Gather_Acc(vertex v, msg_acc, msg)
6   msg_acc.ans ← simd_Or (msg_acc.ans, msg.ans);
7   return msg_acc;

8 Function Apply(vertex v, msg_acc)
9   changed ← simd_AndNot (v.ans, msg_acc.ans); // newly
   reached lanes: ¬v.ans ∧ msg_acc.ans
10  v.ans ← simd_Or (v.ans, msg_acc.ans);

   Function Scatter(vertex v)
11  if NOT simd_All_Zero (changed) then // if changed is True for
   some source
12    msg.ans ← simd_Set (msg.ans, v.ans);
13    Signal (msg);

```

able (e.g., I_t), Codegen uses it directly as the mask for that branch, enabling correct per-lane control flow.

Example 2: Continuing Example 1, in the code-generation step, these labels are interpreted as SIMD masks: T becomes **1** in `Gather/Gather_Acc` and `msg_acc.track` in `Apply/Scatter`, and signaling sets `msg.track` to the current mask. This yields Algorithm 2, where updates to `ans` and `visited` apply only to sources that actually activate the vertex in that round. $\square$

Correctness. The original full paper [41] proves that AutoMI is semantics preserving: for any graph and any set of sources, the generated program $\mathbf{A}_m$ produces exactly the per-source results of running $\mathbf{A}$ independently for each source. Intuitively, `track` ensures that a lane is updated at a vertex if and only if that source would have activated the vertex in a single-instance execution; the mask refinement at control flow further ensures that each lane follows the same branch decisions as in the original program.

Remark. Our rules presented here are for loop-free GAS functions. This matches most vertex programs in practice, as `for` and `while` loops are rarely used in GAS programs since they are essentially local functions running concurrently on each vertex. We remark that one can extend its coverage by adding rules to support loop constructs e.g., `for` and `while`.

4. DROP MASKS (AND GO FASTER)?

Although `track`-guarded vectorization (Section 3) is always correct, it is not free: `track` enlarges messages, adds aggregation work in `Gather_Acc`, and turns many SIMD operations into masked SIMD. For some algorithms, this overhead is avoidable: even if we “over-activate” a vertex for sources that have not reached it yet, the extra executions do not change the final fixed point. We call such programs **TrackFree**.

TrackFree by example: Boolean BFS. Algorithm 3 is a streamlined multi-instance Boolean BFS: each vertex stores a bit-vector `ans` encoding reachability from k sources, aggregates neighbor reachability via $\vee$, and signals only when some lane changes. Notably, the program has *no* `track` field and performs no per-lane activation masking. It is nevertheless correct as reachability is monotone: executing `Apply` on a lane “too early” cannot turn a reachable vertex into unreachable, and repeated applications converge to the same fixed point.

What does it mean to be TrackFree? Consider a single-instance GAS program $\mathbf{A}$. Let $\mathbf{A}_m$ be the general multi-instance program that AutoMI generates in Section 3, which maintains `track` and guards SIMD updates. Let $\mathbf{A}_m^{\text{TF}}$ be the streamlined version that drops `track`, derived from $\mathbf{A}$ via the populate-and-vectorize transformation described in Section 2. We say $\mathbf{A}$ is **TrackFree** if, for any graph G and any set of sources, $\mathbf{A}_m^{\text{TF}}$ returns exactly the same per-source answers as $\mathbf{A}_m$ (which is equivalent to running $\mathbf{A}$ independently for each source). When $\mathbf{A}$ is **TrackFree**, AutoMI can safely drop masks and generate simpler and faster vectorized code.

Determining whether $\mathbf{A}$ is **TrackFree** is subtle: Section 2 shows that Integer BFS fails under streamlined vectorization, while other formulations of the same task (e.g., Bellman-Ford on unweighted graphs) do not. We therefore develop a principled *sufficient* condition of **TrackFree** based on an abstract representation of the program, which allows us to decide **TrackFree** in a unified way rather than reasoning about all possible activation patterns case by case for individual graph algorithms and their GAS programs.

An algebraic characterization. The key difficulty in deciding **TrackFree** is that correctness depends on subtle interactions between (i) the per-round `Apply` update rule and (ii) the engine’s activation schedule. Rather than reasoning about all possible interleavings case-by-case, AutoMI uses an algebraic abstraction of the program semantics. At a high level, we recast the one-round behavior of $\mathbf{A}$ into an abstract `Apply` update function $e_{\mathbf{A}}$ (parameterized by the accumulated message λ). **TrackFree** vectorization corresponds to executing `Apply` on *extra* lanes (either too early or when they are actually inactive). The characterization identifies when such extra executions are harmless by reducing to two idempotence conditions on $e_{\mathbf{A}}$ plus a mild regularity on `Scatter`.

Recasting one GAS round. To reason about when masks can be dropped, AutoMI abstracts a GAS program $\mathbf{A}$ by its *per-round* behavior, separating (i) how vertex state is updated in `Apply` from (ii) how vertices are activated in `Scatter`.

Assume $|V| = n$ and that $\mathbf{A}$ maintains p vertex properties. We view each property as a length- n column vector: for $j \in [1, p]$, $x_j[v]$ is the value of property j at vertex v , and we write the whole state as $\mathbf{x} = (x_1, \dots, x_p)$. In one superstep, `Gather/Gather_Acc` compute an *accumulated message* for each vertex by aggregating information from neighbors. We denote this accumulated message (or tuple of message fields) by λ . In many graph kernels, λ is expressible using the adjacency matrix A of G , e.g., a matrix-vector product such as $\lambda = A^T \cdot \mathbf{x}$ under a suitable semiring; for simplicity, here we treat λ abstractly as “the aggregated input to `Apply`”.

Fix an accumulated message value λ . The `Apply` phase deterministically computes the next vertex state from the current one, using only local state and λ . We denote by $e_{\mathbf{A}}(\cdot)(\lambda)$ the (vector-valued) function induced by `Apply` that maps the current state $\mathbf{x}$ to the state after one round of `Apply` execution when the accumulated message is λ :

$$\mathbf{x}' = e_{\mathbf{A}}(\mathbf{x})(\lambda).$$

Similar to λ , $e_{\mathbf{A}}$ is expressible in matrix languages with a suitable semiring of choice (see the full paper [41] for details). The angle brackets emphasize that λ is treated as a parameter (held fixed), which will be used by the characterization.

Two idempotence conditions. Below we characterize **TrackFree** of $\mathbf{A}$ by relating simple properties of $e_{\mathbf{A}}$ to the correct-

ness of dropping **track** in the vectorized program.

Specifically, **TrackFree** corresponds to executing some **Apply** calls that are redundant (too early, too often, or for lanes that are not truly active). These extra calls are harmless if **Apply** is idempotent in two complementary senses.

(I1) *Active idempotence*. For any fixed accumulated message λ , applying the same update twice has no additional effect:

$$e_A(\mathbf{x})(\lambda) = e_A(e_A(\mathbf{x})(\lambda))(\lambda).$$

Intuitively, this condition means that if a vertex is scheduled multiple times under an unchanged incoming message, the second (and later) applications do not further change state.

(I2) *Inactive idempotence*. Let α be the “inactive” message induced by initialization for a vertex that has not been reached by any source (e.g., $+\infty$ for shortest paths, *False* for reachability). Then executing **Apply** with α must be a no-op:

$$\mathbf{x} = e_A(\mathbf{x})(\alpha).$$

This captures the key requirement for over-activation vertices during GAS rounds: running **Apply** for lanes that are not supposed to be active should not change their state.

Regular signaling. We say that **Scatter** is *regular* if either (a) a vertex always signals neighbors, or (b) it signals only when its state changes in this round (**changed** is *True*). These are common patterns in GAS traversals and iterative kernels, and they allow the algebraic reasoning to connect “who is active” with “who receives a nontrivial message.”

Theorem 1: For a GAS program **A**, if its **Apply** abstraction e_A is both active and inactive idempotent, and its **Scatter** is regular, then **A** is **TrackFree**. $\square$

Applications through examples. Theorem 1 becomes intuitive once we instantiate e_A for familiar kernels. For Boolean BFS, the only state is reachability x , the accumulated message λ is the disjunction of neighbors’ reachability, and **Apply** is $e_A(x)(\lambda) = x \vee \lambda$. Condition (I1) holds as $(x \vee \lambda) \vee \lambda = x \vee \lambda$, and (I2) holds because the inactive message is $\alpha = \text{False}$ and $x \vee \text{False} = x$. Moreover, **Scatter** is regular since it signals only when **changed** is true (Algorithm 3).

For Bellman-Ford/SSSP, **Apply** is $e_A(x)(\lambda) = \min(x, \lambda)$, where $\alpha = +\infty$ for unreachable vertices; $\min$ is idempotent and has identity $+\infty$, and **Scatter** again follows the common “signal if and only if **changed**” pattern and is thus regular. Hence, these programs are **TrackFree**, which also explains why Bellman-Ford computes BFS depths correctly on unweighted graphs under streamlined vectorization. In contrast, Integer BFS violates (I2): even with an “inactive” α , **Apply** can still set **visited** to *True*, changing later branch decisions and signaling, thereby breaking correctness.

Remark. When users believe their algorithm may be **TrackFree**, they provide an algebraic abstraction of **Apply** (capturing the per-round update and the signaling pattern). **AutoMI** then checks the idempotence conditions; if they hold, it generates track-free streamlined program, otherwise it falls back to the general track-guarded compilation of Section 3.

5. APPLICATIONS

We have already seen both sides of **AutoMI** via BFS: Integer BFS needs **track** to be correct (Section 2), while Boolean BFS is **TrackFree** and can drop masks entirely (Section 4). We now highlight additional algorithms that **AutoMI** con-

ALGORITHM 4: **AutoMI** generated multi-instance Bellman-Ford(SSSP)

```

1 #define vertex_data { array <int> ans };
2 #define msg { array <int> ans };

Function Gather(vertex u, edge (u, v), vertex v)
3   msg.ans ← simd_Add (msg.ans, w(u, v));
4   return msg;

5 Function Gather_Acc(vertex v, msg_acc, msg)
6   msg_acc.ans ← simd_Min (msg_acc.ans, msg.ans);
7   return msg_acc;

8 Function Apply(vertex v, msg_acc)
9   mask ← simd_CmpGt (v.ans, msg_acc.ans);
10  v.ans ← simd_mask_Set (mask, v.ans, msg_acc.ans);
11  changed ← simd_mask_Set (mask, changed, 1);

12 Function Scatter(vertex v)
13   if NOT simd_All_Zero (changed) then // if changed is True for
      some source
14     msg.ans ← simd_Set (msg.ans, v.ans);
15     Signal (msg); // activates v's outward neighbors

```

verts, and show how the **TrackFree** characterization of Section 4 explains when streamlined vectorization is safe.

Throughout this section we reuse the notation from Section 4: λ denotes the accumulated (per-vertex) message input to **Apply**, α denotes the corresponding “inactive” message value, and e_A is the **Apply** update function.

SSSP: Bellman-Ford is TrackFree (and subsumes BFS on unweighted graphs). Algorithm 4 shows multi-instance Bellman-Ford program generated by **AutoMI** for SSSP. Each vertex maintains an array $v.\text{ans}$ storing tentative distances from all sources; initially, $v.\text{ans}[i]$ is 0 iff $v = s_i$ and $+\infty$ otherwise. In each round, **Gather** adds edge weights to neighbors’ distances and **Gather_Acc** takes the minimum; **Apply** then performs the standard relaxation (update if smaller distance is found). **Scatter** signals only when some lane changed.

This program is **TrackFree** because its update rule is idempotent and **Scatter** is regular. In our algebraic abstraction, the accumulated message λ at a vertex is the best candidate distance coming from its neighbors, which is expressible as a matrix-vector product under the tropical semiring:

$$\lambda = A^T \cdot x \quad (\oplus = \min, \otimes = +),$$

where x corresponds to the current distance vector $\mathbf{x}_{\text{ans}}$. The **Apply** update is expressed simply as:

$$e_{\text{SSSP}}(x)(\lambda) = \min(x, \lambda).$$

Now both idempotence conditions from Section 4 hold: (a) its active idempotence since, for fixed λ , $\min(\min(x, \lambda), \lambda) = \min(x, \lambda)$; and (b) its inactive idempotence because the inactive message is $\alpha = +\infty$; hence $\min(x, +\infty) = x$.

In addition, its **Scatter** is regular since a vertex only signals when its state changes in the round (i.e., **changed** is *True*). By Theorem 1, SSSP is **TrackFree**, and **AutoMI** safely emits the track-free streamlined code in Algorithm 4. On unweighted graphs, Bellman-Ford computes BFS depths, which also explains the remark from Section 2: BFS *can* be safely streamlined, but only for Boolean queries with an algorithm whose update rule satisfies the idempotence test.

Greedy graph coloring. Algorithm 5 shows **AutoMI**’s multi-instance greedy graph coloring [15]. For each source s_i , it computes an independent coloring by iteratively assigning each vertex the minimum color not used by its already-colored neighbors. The algorithm encodes colors compactly

ALGORITHM 5: AutoMI multi-instance Graph Coloring (GC)

```

1 #define vertex_data { array <bitvec> ans };
2 #define msg { array <bitvec> ans };

3 Function Gather(vertex u, edge (u, v), vertex v)
4   return msg;

5 Function Gather_Acc(vertex v, msg_acc, msg)
6   msg_acc.ans ← simd_Or(msg_acc.ans, msg.ans); // agg
   neighbors' colors
7   return msg_acc;

8 Function Apply(vertex v, msg_acc)
9   mask1 ← simd_CmpNeq(msg_acc.ans, 0); // filter neighbors
   already colored
10  temp ← simd_mask_Add(mask1, msg_acc.ans, 1);
11  c ← simd_mask_AndNot(mask1, temp, msg_acc.ans);
   // min. avail. color
12  mask2 ← simd_mask_CmpNeq(mask1, v.ans, c); // sources
   for new v's color
13  v.ans ← simd_mask_Set(mask2, v.ans, c); // update v's
   color
14  changed ← simd_mask_Set(mask2, changed, 1)

15 Function Scatter(vertex v)
16  if NOT simd_All_Zero(changed) then // if changed is True for
   some source
17    msg.ans ← simd_Set(msg.ans, v.ans); // send out v's
   color for next rd.
18    Signal(msg); // activates v's outward neighbors & send msg

```

as bit-vectors: a bit-vector value y indicates which colors appear in the neighborhood, and the minimum available color can be computed by a standard bit trick $(y + 1) \& \sim y$.

Despite the more involved **Apply** logic, the same **TrackFree** reasoning applies. Let x denote the per-vertex color-state vector (bit-vectors) and let λ denote the aggregated neighbor color-state, computed via bitwise OR:

$$\lambda = A^T \cdot x \quad (\oplus = \text{OR}, \otimes = \times).$$

Apply is abstracted as $e_{GC}(x)(\lambda) = h(x, \lambda)$, where h returns x when no neighbor is colored ($\lambda = 0$), and otherwise returns minimum available color derived from λ (as in Algorithm 5).

Two facts make it **TrackFree**: (i) for fixed λ , the value computed by $h(x, \lambda)$ is stable after one application, so reapplying with the same λ has no further effect (active idempotence); and (ii) the inactive message is $\alpha = 0$, so $h(x, 0) = x$ (inactive idempotence). Finally, the program signals only on updates, which implies that **Scatter** is regular. Therefore, graph coloring satisfies Theorem 1, and **AutoMI** can again drop **track** and generate streamlined vectorized code.

Advanced graph analytics: SpMV, PPR, and CF. Beyond traversals, **AutoMI** supports analytics kernels that are naturally written as vertex-centric iterative updates, including Sparse Matrix-Vector multiplication (SpMV), Personalized PageRank (PPR), and Collaborative Filtering (CF).

SpMV and PPR (TrackFree). We follow [20, 31, 14, 34, 22] to reduce SpMV to a graph traversal problem; similarly for PPR as an extension of SpMV. Specifically, SpMV computes an output vector $y = Ax$ (one value per vertex) by aggregating neighbor contributions; PPR performs repeated SpMV-style propagation with a source-specific restart and personalization distribution [28]. In GAS implementations of these kernels, **Apply** typically overwrites the current per-vertex value with a function of the accumulated message (and possibly a per-source constant, for restart). As a result, for fixed λ the **Apply** update is stable after one execution (satisfying (I1)), and the inactive message corresponds to the aggregation identity (satisfying (I2)). Moreover, their **Scatter** fol-

Table 3: Average speedup of AutoMI, Quegel and MultiLyra over PowerLyra (averaged over LiveJournal [5], Twitter [8], Friendster [1], UKDomain [9], MovieLens [6], Netflix [7], up to 3 billion of edges; $\times$ marks algorithms not available.)

system	bBFS	iBFS	SSSP	GC	SpMV	PPR	CF
PowerLyra [13]	1	1	1	1	1	1	1
Qegel [37, 40]	2.32x	2.25x	1.77x	$\times$	$\times$	$\times$	$\times$
MultiLyra [26]	6.07x	5.53x	5.16x	$\times$	$\times$	$\times$	$\times$
AutoMI	14.82x	9.51x	14.09x	6.37x	7.79x	8.21x	3.78x

lows a regular pattern, *i.e.*, signal-on-change. Hence, SpMV and PPR are **TrackFree** under the criterion, and **AutoMI** can generate streamlined vectorized multi-instance code.

Collaborative filtering (not TrackFree). In contrast, CF uses Stochastic Gradient Descent [19] to iteratively update latent factors of vertices being visited, starting from a source. Its incremental parameter updates are however not idempotent: repeating **Apply** with the same incoming signal continues to change latent factors, violating (I1). Therefore **TrackFree** is unsafe for CF, and **AutoMI** falls back to the general track-guarded compilation from Section 3 to preserve correctness.

Takeaway. These examples highlight **AutoMI**’s practical impact: without changing the underlying GAS engine, **AutoMI** compiles existing single-instance vertex-centric programs into efficient multi-instance SIMD implementations. When misaligned activations would break correctness, **AutoMI** automatically injects and propagates **track** to enforce per-source semantics (Section 3); when the idempotence conditions of Section 4 hold, it drops masks entirely and emits streamlined code (**TrackFree**) for even higher performance.

In short, developers can write (or reuse) standard GAS algorithms once, while **AutoMI** decides how to vectorize them *safely* and delivers multi-source speedups without hand-engineered multi-instance implementations.

6. EXPERIMENTAL STUDY

We summarize the key experimental findings of the full paper [41], focusing on (i) end-to-end speedups over state-of-the-art distributed vertex-centric baselines, and (ii) the practical contribution of SIMD and **TrackFree**.

6.1 Implementation and Evaluation Setup

System. **AutoMI** is a source-to-source compiler for C++ GAS programs. It uses libclang [4] to parse a single-instance program into an AST, applies the annotation rules of Section 3, and then emits a compilable multi-instance C++ program by populating vertex/message fields into length- k arrays and vectorizing the resulting operations. When the **TrackFree** criterion of Section 4 holds, **AutoMI** generates the streamlined variant that omits **track** and uses unmasked SIMD. The generated code includes the populated data types and their serialization routines (needed for distributed message passing), and runs on standard GAS engines without runtime changes (**PowerLyra** [13] by default).

To make SIMD effective, **AutoMI** stores populated property contiguously with aligned allocation, and maps common operators to Intel SIMD intrinsics [3]; unsupported cases fall back to a small “abstract SIMD” loop (*e.g.*, bit-vector operations in greedy coloring are applied over 256-bit chunks).

Compared systems. We compare against **PowerLyra** [13] (serially running the single-instance program concurrently over

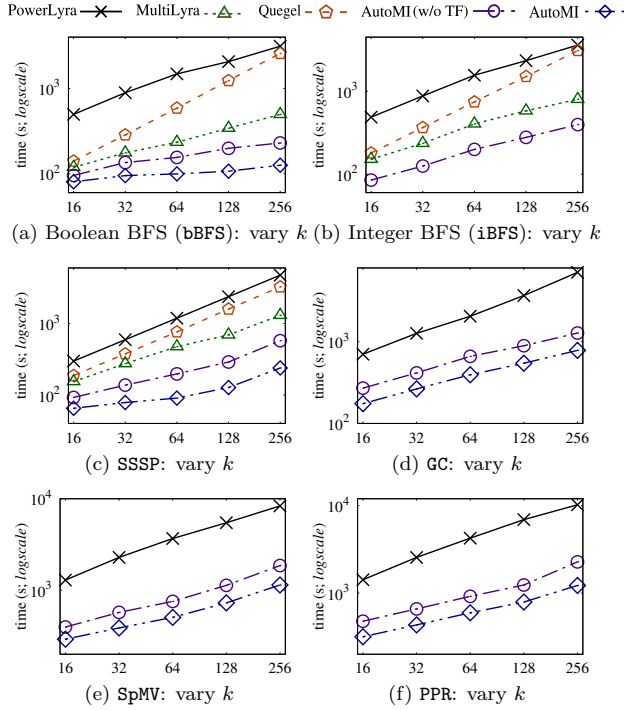


Figure 3: Running time of multi-instance bBFS, iBFS, SSSP, GC, SpMV, PPR and CF with varying number (k) of queries. Results are over *Friendster* [1] graph with 68,349,466 vertices and 2,586,147,869 edges; AutoMI(w/oTF) denotes AutoMI without *TrackFree* optimization, *i.e.*, always using *track*.

all threads, once per source), MultiLyra [26] (authors’ handwritten multi-instance BFS/SSSP atop the same PowerLyra engine), and Quegel [37, 40] (batch query processing; adapted to point-to-all evaluation). Since AutoMI, PowerLyra, and MultiLyra share the same engine, their comparison isolates the effect of code generation, SIMD, and *TrackFree*; Quegel serves as a representative batching baseline.

Workloads and configuration. We use six real graphs, up to 3.3 billion edges (see [41] for details). For SSSP, edge weights are sampled uniformly from $[1, \log |V|]$. For each graph, we generate $k \in \{16, 32, 64, 128, 256\}$ sources by sampling a seed vertex and selecting k vertices reached by a BFS from the seed; results are averaged over three seeds. We evaluate bBFS, iBFS, and SSSP on all systems; GC, SpMV, PPR, and CF are evaluated with AutoMI and PowerLyra only (not available in MultiLyra/Quegel). All experiments run on four AWS r6i.8xlarge instances (32 vCPUs, 256GB RAM each) connected by a 12.5Gbps network.

6.2 Performance of Vectorized Programs

Overall performance. Table 3 reports the average speedups over PowerLyra across all graphs for each system, and Fig. 3 depicts the detailed evaluation time of all methods over billion-edge graph *Friendster* [1] with varying number k of instances. The results confirm that AutoMI is effective, and is consistently the best among all for each and every case tested. As shown in Table 3, multi-instance GAS programs produced by AutoMI significantly outperform the serial evaluation by PowerLyra, highly optimized MultiLyra code and

Table 4: Run time speedup over PowerLyra, breakdown and #comms for SSSP over UKDomain [9] with 256 queries

system	speedup	Gather (s)	Apply (s)	Scatter (s)	Sync (s)	#comms
PowerLyra	1	1787.14	1324.96	2033.49	304.67	4.61×10^{10}
MultiLyra	3.07x	801.48	777.87	183.55	12.25	4.73×10^8
AutoMI _{w/o TF}	5.46x	438.89	416.98	139.79	12.24	4.73×10^8
AutoMI	11.32x	192.84	196.16	83.68	11.81	4.73×10^8

Quegel batch evaluation for all cases. It is up to $27.83\times$, $16.42\times$, $29.51\times$, $9.84\times$, $10.26\times$, $10.65\times$ and $6.17\times$ faster than PowerLyra for bBFS, iBFS, SSSP, GC, SpMV, PPR and CF, respectively. Compared to MultiLyra and Quegel, AutoMI is up to $3.82\times$ and $26.35\times$ faster for bBFS, $2.57\times$ and $7.12\times$ for iBFS, $4.63\times$ and $23.01\times$ for SSSP.

Breakdown. To isolate where AutoMI gains its speedups, Table 4 reports a per-phase breakdown for SSSP on UKDomain [9] with 256 sources, together with the total number of messages (#comms). We also report AutoMI_{w/o TF}, the AutoMI variant that always using the generic *track*-guarded vectorization.

All multi-instance methods sharply reduce communication: #comms drops from 4.61×10^{10} (PowerLyra, serial) to 4.73×10^8 , and sync time shrinks from 304.67s to about 12s. With the same #comms as MultiLyra, AutoMI (w/o TF) is $1.78\times$ faster overall, highlighting the benefit of SIMD. Enabling *TrackFree* removes *track* overhead and yields a further $2.07\times$ over AutoMI (w/o TF), giving $11.32\times$ speedup over PowerLyra (and $3.69\times$ over MultiLyra).

These reveal three sources of performance gains of AutoMI: (1) *multi-instance interleaving* reduces redundant traversal and communication; (2) *SIMD vectorization* accelerates per-vertex computation relative to handwritten multi-instance code (*e.g.*, MultiLyra); and (3) for *TrackFree* kernels, dropping *track* removes the mask maintenance overhead.

7. CONCLUSION

Multi-source graph workloads are ubiquitous, yet SIMD-accelerated multi-instance implementations are notoriously hard to write correctly, especially in vertex-centric distributed environment. AutoMI makes this practical: it automatically compiles a standard single-instance GAS program into a correct multi-instance SIMD program by propagating a lightweight per-message mask (*track*) and using it to guard vectorized updates. Moreover, AutoMI goes beyond “always-mask” compilation: its *TrackFree* optimization and algebraic characterization identify when masks can be dropped entirely, yielding streamlined code with substantially higher performance. Putting together, AutoMI turns multi-instance algorithm development and SIMD efficiency from a manual engineering burden into a compiler-driven capability.

A natural next step is to have broader language and run-time support. While many GAS programs are loop-free and fit the current design, supporting richer control flow, additional vertex-centric interfaces beyond GAS, and alternative execution semantics (*e.g.*, asynchronous) would broaden applicability. Another interesting direction is to target SIMT architectures. This requires rethinking the compilation strategy to generate GPU kernels. A particularly interesting goal is an analogue of *TrackFree* for GPUs: characterizing when a multi-instance program can be safely executed with relaxed per-source scheduling or with cheaper warp masking while still converging to the correct fixed point.

8. REFERENCES

- [1] Friendster. <http://konect.cc/networks/friendster/>, Access: 2024.
- [2] Giraph. <https://giraph.apache.org/>, Access: 2024.
- [3] Intel® intrinsics guide. <https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html>, Access: 2024.
- [4] libclang. <https://clang.llvm.org/>, Access: 2024.
- [5] LiveJournal. <http://snap.stanford.edu/data/soc-LiveJournal1.html>, Access: 2024.
- [6] MovieLens. <http://grouplens.org/datasets/movielens/>, Access: 2024.
- [7] Netflix. <http://konect.cc/networks/netflix/>, Access: 2024.
- [8] Twitter. <http://konect.cc/networks/twitter/>, Access: 2024.
- [9] UK domain. <http://konect.cc/networks/dimacs10-uk-2007-05/>, Access: 2024.
- [10] Z. Abul-Basher. Multiple-query optimization of regular path queries. In *33rd IEEE International Conference on Data Engineering, ICDE 2017, San Diego, CA, USA, April 19-22, 2017*, pages 1426–1430. IEEE Computer Society, 2017.
- [11] K. Ammar and M. T. Özsu. Experimental analysis of distributed graph systems. *Proc. VLDB Endow.*, 11(10):1151–1164, 2018.
- [12] N. Bruno, L. Gravano, N. Koudas, and D. Srivastava. Navigation- vs. index-based XML multi-query processing. In U. Dayal, K. Ramamritham, and T. M. Vijayaraman, editors, *Proceedings of the 19th International Conference on Data Engineering, March 5-8, 2003, Bangalore, India*, pages 139–150. IEEE Computer Society, 2003.
- [13] R. Chen, J. Shi, Y. Chen, B. Zang, H. Guan, and H. Chen. PowerLyra: Differentiated graph computation and partitioning on skewed graphs. *ACM Transactions on Parallel Computing (TOPC)*, 5(3):1–39, 2019.
- [14] M. K. Esfahani, P. Kilpatrick, and H. Vandierendonck. Exploiting in-hub temporal locality in spmv-based graph processing. In X. Sun, S. Shende, L. V. Kalé, and Y. Chen, editors, *ICPP 2021: 50th International Conference on Parallel Processing, Lemont, IL, USA, August 9 - 12, 2021*, pages 42:1–42:10. ACM, 2021.
- [15] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *10th USENIX symposium on operating systems design and implementation (OSDI 12)*, pages 17–30, 2012.
- [16] M. Han, K. Daudjee, K. Ammar, M. T. Özsu, X. Wang, and T. Jin. An experimental comparison of pregel-like graph processing systems. *Proc. VLDB Endow.*, 7(12):1047–1058, 2014.
- [17] U. Kang, C. E. Tsourakakis, and C. Faloutsos. PEGASUS: mining peta-scale graphs. *Knowl. Inf. Syst.*, 27(2):303–325, 2011.
- [18] M. Kaufmann, M. Then, A. Kemper, and T. Neumann. Parallel array-based single- and multi-source breadth first searches on large dense graphs. In V. Markl, S. Orlando, B. Mitschang, P. Andritsos, K. Sattler, and S. Breß, editors, *Proceedings of the 20th International Conference on Extending Database Technology, EDBT 2017, Venice, Italy, March 21-24, 2017*, pages 1–12. OpenProceedings.org, 2017.
- [19] Y. Koren, R. Bell, and C. Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 42(8):30–37, 2009.
- [20] A. Kyrola, G. E. Blelloch, and C. Guestrin. GraphChi: Large-scale graph computation on just a PC. In C. Thekkath and A. Vahdat, editors, *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, pages 31–46. USENIX Association, 2012.
- [21] W. Le, A. Kementsietsidis, S. Duan, and F. Li. Scalable multi-query optimization for SPARQL. In A. Kementsietsidis and M. A. V. Salles, editors, *IEEE 28th International Conference on Data Engineering (ICDE 2012), Washington, DC, USA (Arlington, Virginia), 1-5 April, 2012*, pages 666–677. IEEE Computer Society, 2012.
- [22] J. Li, W. Zhao, N. Ntarmos, Y. Cao, and P. Buneman. MITra: A framework for multi-instance graph traversal. *Proceedings of the VLDB Endowment*, 16(10), 2023.
- [23] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment*, 5(8), 2012.
- [24] Y. Lu, J. Cheng, D. Yan, and H. Wu. Large-scale distributed graph computing systems: an experimental evaluation. *Proc. VLDB Endow.*, 8(3):281–292, 2014.
- [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 A. K. Elmagarmid and D. Agrawal, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, pages 135–146. ACM, 2010.
- [26] A. Mazloumi, X. Jiang, and R. Gupta. MultiLyra: Scalable distributed evaluation of batches of iterative graph queries. In *2019 IEEE International Conference on Big Data (Big Data)*, pages 349–358. IEEE, 2019.
- [27] R. R. McCune, T. Weninger, and G. Madey. Thinking like a vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Computing Surveys (CSUR)*, 48(2), 2015.
- [28] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.
- [29] P. Peng, Q. Ge, L. Zou, M. T. Özsu, Z. Xu, and D. Zhao. Optimizing multi-query evaluation in federated RDF systems. *IEEE Trans. Knowl. Data Eng.*, 33(4):1692–1707, 2021.
- [30] X. Ren and J. Wang. Multi-query optimization for subgraph isomorphism search. *Proc. VLDB Endow.*,

- 10(3):121–132, 2016.
- [31] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 472–488, 2013.
 - [32] T. K. Sellis. Multiple-query optimization. *ACM Trans. Database Syst.*, 13(1):23–52, 1988.
 - [33] T. K. Sellis and S. Ghosh. On the multiple-query optimization problem. *IEEE Trans. Knowl. Data Eng.*, 2(2):262–266, 1990.
 - [34] J. Sun, H. Vandierendonck, and D. S. Nikolopoulos. Accelerating graph analytics by utilising the memory locality of graph partitioning. In *2017 46th International Conference on Parallel Processing (ICPP)*, pages 181–190. IEEE, 2017.
 - [35] M. Then, M. Kaufmann, F. Chirigati, T. Hoang-Vu, K. Pham, A. Kemper, T. Neumann, and H. T. Vo. The more the merrier: Efficient multi-source graph traversal. *Proc. VLDB Endow.*, 8(4):449–460, 2014.
 - [36] D. Yan, Y. Bu, Y. Tian, A. Deshpande, et al. Big graph analytics platforms. *Foundations and Trends® in Databases*, 7(1-2):1–195, 2017.
 - [37] D. Yan, J. Cheng, M. T. Özsu, F. Yang, Y. Lu, J. C. S. Lui, Q. Zhang, and W. Ng. A general-purpose query-centric framework for querying big graphs. *Proc. VLDB Endow.*, 9(7):564–575, 2016.
 - [38] H. Yanagisawa. A multi-source label-correcting algorithm for the all-pairs shortest paths problem. In *24th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010, Atlanta, Georgia, USA, 19-23 April 2010 - Conference Proceedings*, pages 1–10, 2010.
 - [39] Q. Zhang, H. Chen, D. Yan, J. Cheng, B. T. Loo, and P. Bangalore. Architectural implications on the performance and cost of graph analytics systems. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 40–51, 2017.
 - [40] Q. Zhang, D. Yan, and J. Cheng. Quegel: A general-purpose system for querying big graphs. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD ’16*, pages 2189–2192. ACM, 2016.
 - [41] W. Zhao, Y. Cao, P. Buneman, J. Li, and N. Ntarmos. Automating vectorized distributed graph computation. *Proc. ACM Manag. Data*, 2(6):254:1–254:27, 2024.
 - [42] W. Zhao, J. Li, Y. Cao, and N. Ntarmos. Mitra: Populating graph traversal algorithms. In *ICDE*, pages 4576–4579. IEEE, 2025.

Technical Perspective: Output-Optimal Evaluation of Conjunctive Queries

Dan Suciu
University of Washington
suciu@cs.washington.edu

The best asymptotic runtime that we can hope for a query evaluation algorithm is $O(N + \text{OUT})$, where N is the size of the input database and OUT is the size of the query's output. Indeed, any algorithm must read the input, and must write the output, hence one cannot do better asymptotically. It is not immediately obvious how to evaluate a query in this time. The standard approach taken by all query engines is to compute one join at a time, but in that case the intermediate results may become much larger than both the input and the final output. Yannakakis' algorithm from the early 80's achieves a runtime of $O(N + \text{OUT})$ on all full, acyclic queries, by first removing all dangling tuples from all relations using a sequence of semijoins, then performing the joins.

The vast majority of SQL queries found in real workloads are acyclic, so the restriction of Yannakakis' algorithm to acyclic queries is not an important limitation. However, the restriction to *full* queries means that its runtime can only be guaranteed for `select * from ...` queries. In particular, Yannakakis' algorithm does not apply to queries with a `group by` clause, which form the backbone of day to day data analytics.

Most query engines compute a query with a `group by` clause by first performing all joins, then applying the group-by-aggregate operator at the end. In other words, they compute the full join, and compute the `group by` at the end. But the size of the full join is often much larger than the number of groups in the final output, which means that the runtime can be arbitrarily larger than both the input and the output.

A few query engines support an alternative approach: they push the group-by-aggregate operator down the query plan, thus aiming to reduce the size of the intermediate results. But even if one removes all dangling tuples before performing the joins, pushing group-by-aggregate down the query plan can only guarantee a runtime of $O(N \cdot \text{OUT})$, which is a sharp increase over our desired $O(N + \text{OUT})$. The only exception are so called *acyclic, free-connex* queries. These are queries that satisfy a stringent syntactic restriction, where all `group by` variables form a connected component in the join tree. The only `group by` queries that can be evaluated in time $O(N + \text{OUT})$ are the acyclic, free-connex queries: these can be computed with this runtime by pushing the group-by-aggregate operators down the join tree.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to publish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright 2008 ACM 0001-0782/08/0X00 ...\$5.00.

The following paper, by Hu et al., describes an algorithm that reduces the complexity of query evaluation for any acyclic query to $O(N + N \cdot \text{OUT}^{1-\varepsilon} + \text{OUT})$, where $0 < \varepsilon \leq 1$ is a query-dependent constant. The running example in the paper is a chain query with two output variables (group by variables), where this constant is 2, and therefore the runtime is $O(N + N \cdot \text{OUT}^{1/2} + \text{OUT})$. This runtime is better than $O(N \cdot \text{OUT})$, without reaching the ideal $O(N + \text{OUT})$. When the query happens to be free-connex, then $\varepsilon = 1$ and the algorithm has the same runtime as Yannakakis' algorithm.

The paper proves, in Theorem 4, that this algorithm is optimal. The statement of the theorem is somewhat technical: it proves that the circuit needed to represent the query's output has size $\Omega(N \cdot \text{OUT}^{1-\varepsilon})$, where ε is the same as in the algorithm. This implies that the algorithm presented in the paper has the best possible runtime among all "combinatorial" algorithms, understood here as those algorithm whose trace is a circuit (this precludes, e.g., algorithms using fast matrix multiplication).

So how does one compute an acyclic query in this optimal runtime? The algorithm is not straightforward, but the paper introduces it gently, by illustrating it on a chain query. At its core, the algorithm is based on partitioning each input relation into *light* and *heavy* tuples, then computing the query separately on the various combinations of light and heavy relations. This technique has been applied in the algorithm literature to various graph pattern matching algorithms, and it is also popular in distributed systems (for example *SkewJoin* hash-partitions the light elements, and broadcasts the heavy elements). One difference in the current paper is that the definition of light and heavy tuples is based on the size of the output, which needs to be computed (approximatively) in a separate step.

Finally, is this new algorithm ready for production? Not yet. Research, at its best, evolves from theory to practical implementation. Perhaps the best illustration of such an evolution is Yannakakis' algorithm itself, which was introduced in 1981, but whose practical adaptation has been investigated only recently. This paper makes a major first step in generalizing Yannakakis' algorithm to all queries with `group-by`, and solves this problem completely and definitively, by providing theoretical matching upper and lower bounds for the query evaluation problem. More, and significant research is still needed to incorporate these ideas into today's highly optimized query engines.

Output-Optimal Evaluation of Conjunctive Queries

Xiao Hu
University of Waterloo
Canada
xiaohu@waterloo.ca

Shaleen Deep
Microsoft
USA
shaleen.deep@microsoft.com

Paraschos Koutris
UW-Madison
Madison, WI, USA
paris@cs.wisc.edu

Austen Z. Fan
UW-Madison
Madison, WI, USA
afan@cs.wisc.edu

Hangdong Zhao
UW-Madison
Madison, WI, USA
hangdong@cs.wisc.edu

ABSTRACT

One of the most celebrated results for evaluating conjunctive queries (CQs) is the Yannakakis algorithm [24] proposed in 1981. It is known that free-connex CQs can be evaluated in $O(N + \text{OUT})$ time, where N is the input size of the database and OUT is the output size of the query result. This is already output-optimal. However, only an upper bound $O(N \cdot \text{OUT})$ on the runtime is known for the remaining acyclic but non-free-connex CQs. Alternatively, one can convert a non-free-connex CQ into a free-connex one using tree decomposition techniques, and then run the Yannakakis algorithm. However, none of them is known to be output-optimal.

In this work, we show a lower and upper bound matching $\Theta\left(N \cdot \text{OUT}^{1 - \frac{1}{\text{fn-fhtw}}} + \text{OUT}\right)$ for computing acyclic CQs, where fn-fhtw is the *free-connex fractional hypertree width* of the query. Although free-connex fractional hypertree width is a natural and well-established measure of how far a CQ is from being free-connex, we demonstrate that it precisely captures the output-optimal complexity of acyclic CQs. To our knowledge, this has been the first polynomial improvement over the Yannakakis algorithm in the last 40 years and completely resolves the open question of computing acyclic CQs in an output-optimal way. Our output-optimal algorithm proposed for acyclic CQs also extends to cyclic CQs, as well as CQs with aggregations over arbitrary commutative semirings.

1. INTRODUCTION

Conjunctive query (CQ) evaluation is a fundamental task for relational databases. Finding efficient algorithms for evaluating CQs has been a holy grail in

database theory since 1981. Previous results have achieved two flavors of runtimes: *worst-case optimal* [21, 18, 15, 24, 4] and *output-sensitive* [24, 15]. Worst-case optimal bounds are tight only on pathological instances with large outputs, which is uncommon in practice. In contrast, output-sensitive bounds express the runtime as a function of the input size N and output size OUT , which are more practically meaningful, especially for queries where the aggregation or projection operator may significantly reduce the output size. In addition, an output-optimal algorithm is also worst-case optimal (but not vice versa): the Yannakakis algorithm [24] is the best-known example, which achieves an output-optimal bound $O(N + \text{OUT})$ for *free-connex* queries.

Although output-optimal algorithms are practically desirable, they are much more difficult to design. For the large class of *acyclic but non-free-connex* queries, prior works have yet to discover an output-optimal algorithm. Currently, there are two predominant approaches to computing these queries: (1) run the Yannakakis algorithm; (2) convert the query into a *free-connex* one using the *tree decomposition* technique and the *worst-case optimal join* algorithm [18, 19], and then run the Yannakakis algorithm. For (1), Yannakakis only gave an upper bound $O(N \cdot \text{OUT})$ on its runtime. Later, this bound was tightened to $O\left(N \cdot \text{OUT}^{1 - \frac{1}{k}}\right)$ for star CQs with k relations, which is output-optimal [20]. For (2), Khamis et al. showed an upper bound $O\left(N^{\#\text{fn-subw}} + \text{OUT}\right)$ on its runtime, where $\#\text{fn-subw}$ is the *#free-connex submodular width* of the query [15]. Both algorithms are worst-case optimal, but neither is known to be *output-optimal* for general acyclic CQs.

In this work, we identify the free-connex fractional hypertree width (fn-fhtw) for CQs to characterize the output-optimal complexity. Specifically, we show how to evaluate acyclic CQs in $\Theta\left(N \cdot \text{OUT}^{1 - \frac{1}{\text{fn-fhtw}}} + \text{OUT}\right)$ time, together with a matching lower bound under the semiring circuit model. Note that $\text{fn-fhtw} = 1$ for the free-connex queries and $\text{fn-fhtw} = k$ for star queries with k relations, thus generalizing the previous results on these two special cases. As a by-product, our output-optimal algorithm for acyclic queries also yields new output-sensitive algorithms for cyclic queries. In ad-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

This is an invitation of two independent works published in Proc. ACM Manag. Vol. 2, No. 5 (PODS), Article 220 (<https://doi.org/10.1145/3695838>), and Vol 3, No. 2 (PODS), Article No. 104 (<https://doi.org/10.1145/3725241>).

dition, all of our results can be extended to aggregation queries over commutative semirings.

2. PRELIMINARIES

We establish the minimal notation and concepts required to state our main result.

Conjunctive Query (CQ). We associate a CQ Q with a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$, where the set of vertices $\mathcal{V} = \{x_1, \dots, x_n\}$ model the *attributes* and the set of hyperedges $\mathcal{E} = \{e_1, \dots, e_k\} \subseteq 2^{\mathcal{V}}$ model the *relations*. Let $\text{dom}(x)$ be the *domain* of attribute $x \in \mathcal{V}$. Let $\text{dom}(X) = \prod_{x \in X} \text{dom}(x)$ be the *domain* of a subset $X \subseteq \mathcal{V}$ of attributes. An *instance* of Q is associated a set of relations $\mathcal{D} = \{R_e(\mathbf{x}_e) : e \in \mathcal{E}\}$, where $\mathbf{x}_e \subseteq \mathcal{V}$ is the schema (or the set of variables) of R_e . each relation R_e consists of a set of *tuples*, where each tuple $t \in R_e$ is an assignment that assigns a value from $\text{dom}(x)$ to x for every attribute $x \in \mathbf{x}_e$. The CQ is written as:

$$Q(\mathbf{x}_{\mathcal{F}}) \leftarrow \bigwedge_{e \in \mathcal{E}} R_e(\mathbf{x}_e)$$

The variables $\mathbf{x}_{\mathcal{F}}$ in the head of the query are the free variables (or the output variables). In this way, a CQ Q can be modeled as a tuple $(\mathcal{H}, \mathcal{F})$. The query result of Q on the instance $\mathcal{D}$, denoted as $Q(\mathcal{D})$, is defined as:

$$Q(\mathcal{D}) = \{t' \in \text{dom}(\mathbf{x}_{\mathcal{F}}) : \exists t \in \text{dom}(\mathcal{V}), \pi_{\mathbf{x}_{\mathcal{F}}} t = t', \\ \forall e \in \mathcal{E}, \pi_{\mathbf{x}_e} t \in R_e\}.$$

We use N to denote the input size of $\mathcal{D}$, and use $\text{OUT} = |Q(\mathcal{D})|$ to denote the output size.

We study the data complexity of this problem by assuming the query size (i.e., $|\mathcal{V}|$ and $|\mathcal{E}|$) is constant. Our objective in this work is to design algorithms for evaluating CQs by taking both the input and output size into account. In other words, the running time we obtain should be a function of both N and OUT .

Fractional Edge Covering Number. For a subset $S \subseteq \mathcal{V}$, we use $\mathcal{H}[S] = (S, \mathcal{E}[S])$ to denote the sub-hypergraph induced by S , where $\mathcal{E}[S] = \{e \cap S : e \in \mathcal{E}\}$. For any subset $S \subseteq \mathcal{V}$, a *fractional edge covering* of S is a function $\rho : \mathcal{E} \rightarrow [0, 1]$ such that $\sum_{e \in \mathcal{E} : x \in e} \rho(e) \geq 1$ for each variable $x \in S$. The *fractional edge covering number* of $\mathcal{H}$, denoted as $\rho^*(\mathcal{H})$, is defined as the minimum sum of weight over all possible fractional edge coverings ρ of $\mathcal{V}$, i.e., $\rho^*(\mathcal{H}) = \min_{\rho} \sum_{e \in \mathcal{E}} \rho(e)$.

Tree Decomposition. A *tree decomposition* (TD) of $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ is a pair $(\mathcal{T}, \chi)$, where $\mathcal{T}$ is a tree and $\chi : \text{nodes}(\mathcal{T}) \rightarrow 2^{\mathcal{V}}$ is a mapping from the nodes of $\mathcal{T}$ to subsets of $\mathcal{V}$, that satisfies the following properties: (1) For each relation $e \in \mathcal{E}$, there is a node $u \in \text{nodes}(\mathcal{T})$ such that $\mathbf{x}_e \subseteq \chi(u)$. (2) For each attribute $A \in \mathcal{V}$, the set $\{u \in \text{nodes}(\mathcal{T}) : A \in \chi(u)\}$ forms a connected sub-tree of $\mathcal{T}$.

Each set $\chi(u)$ is called a *bag* of the TD. The *width* of $(\mathcal{T}, \chi)$ noted as $\text{width}(\mathcal{T}, \chi)$ is defined as

$$\text{width}(\mathcal{T}, \chi) = \max_{u \in \text{nodes}(\mathcal{T})} \rho^*(\mathcal{H}[\chi(u)])$$

i.e., the maximum fractional edge covering number of the subqueries induced by bags in $\mathcal{T}$. A TD $(\mathcal{T}, \chi)$ is

free-connex for a CQ $(\mathcal{H}, \mathcal{F})$ if there is a connected subtree S of $\mathcal{T}$ such that $\bigcup_{u \in \text{nodes}(S)} \chi(u) = \mathbf{x}_{\mathcal{F}}$, i.e., the union of attributes appearing in S is exactly the output variables. Figure 1 shows a tree decomposition rooted at two different nodes.

Classification of CQs. A CQ is acyclic [6, 14] if and only if it has a width-1 TD. A CQ is free-connex [5] if and only if it has a width-1 free-connex TD. As an example, both tree decompositions in Figure 1 have width 1.

Model of Computation. We use the standard RAM model with a uniform cost measure. For an instance of size N , every register has length $O(\log N)$. Any arithmetic operation (such as addition, subtraction, multiplication, and division) on the values of two registers can be done in $O(1)$ time. Sorting the values of N registers can be done in $O(N \log N)$ time.

3. CHAIN CQS

In this section, we first review the classic Yannakakis algorithm, then examine chain queries (a.k.a. chain matrix multiplication):

$$Q_{\text{chain}}(x_1, x_{k+1}) \leftarrow \bigwedge_{i \in [k]} R_i(x_i, x_{i+1}).$$

on which the Yannakakis algorithm is not optimal, and finally, show our optimal algorithm. We will use the 4-chain query Q_{chain4} ($k = 4$) as a running example.

3.1 Yannakakis Algorithm

Suppose we are given an acyclic CQ $Q = (\mathcal{H}, \mathcal{F})$ and an instance $\mathcal{D}$ for Q . For simplicity, we assume that there exists no pair of relations $e, e' \in \mathcal{E}$ such that $e \subseteq e'$; otherwise, we can simply replace $R_{e'}$ by $R_{e'} \bowtie R_e$ and remove R_e . Let $(\mathcal{T}, \chi)$ be a width-1 TD for Q . There is a one-to-one correspondence between relations in $\mathcal{E}$ and nodes in $\mathcal{T}$. For simplicity, we also use e to denote the node u in $\mathcal{T}$ that corresponds to relation e , i.e., $\chi(u) = e$. The algorithm consists of two phases:

Semi-Joins. The first phase removes all *dangling tuples* via a bottom-up and top-down pass of semi-joins along $\mathcal{T}$, where a tuple is *dangling* if it does not participate in any full join result. This step can be performed in linear time $O(N)$.

Join-Project. The second phase roots $\mathcal{T}$ on an arbitrary root node r , and then performs joins and projections in a bottom-up way. Specifically, it takes two nodes R_u and R_{p_u} such that u is a leaf and p_u is the parent of u , projects away non-output variables that do not appear in $\chi(p_u)$ by replacing R_u with $\pi_{\chi(u) \cap (\chi(p_u) \cup \mathbf{x}_{\mathcal{F}})} R_u$, and replaces R_{p_u} with $R_{p_u} \bowtie R_u$. Then R_u is removed, and the step repeats until only one node remains, i.e., the root node r . Hence, it projects on output variables in r , and outputs $\pi_{\mathbf{x}_{\mathcal{F}}} R_r$ as the final result.

Figure 1 shows the query plans of running Yannakakis' algorithm on two rooted tree decompositions.

The runtime is proportional to the largest number of intermediate results materialized (after dangling tuples are removed). It can be shown that this number can always be upper bounded by $O(N \cdot \text{OUT})$, hence the

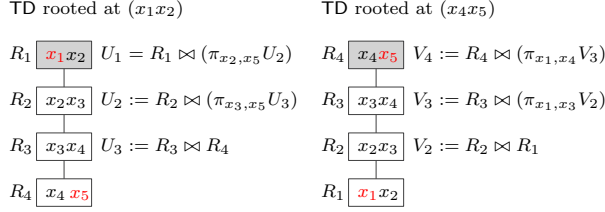


Figure 1: Two width-1 TDs for Q_{chain4} . For TD rooted at (x_1x_2) , U_3, U_2, U_1 are the intermediate results materialized by the Yannakakis algorithm. For TD rooted at (x_4x_5) , V_2, V_3, V_4 are the intermediate results materialized by the Yannakakis algorithm.

above algorithm takes time $O(N + N \cdot \text{OUT})$. When the join query is free-connex, it can be shown that the runtime can be bounded instead by the tighter expression $O(N + \text{OUT})$. However, this upper bound can sometimes be loose and dependent on the specific *query plan* chosen for a given input instance.

3.2 Why Yannakakis is suboptimal

We now construct an instance as shown in Figure 2, where we will show that Yannakakis' algorithm has a suboptimal behavior, independent of which plan we choose. This is a generalization of the analysis in [11].

Example 1. Assume $2 \leq \text{OUT} \leq N$. Let us first focus on the top half of Figure 2). Attributes x_1, x_2, x_3, x_4 have domain sizes $\frac{\text{OUT}}{2}, \frac{N}{\text{OUT}}, \frac{N}{2}, 1$ respectively. R_1 is a Cartesian product between $\text{dom}(x_1)$ and $\text{dom}(x_2)$, R_2 is a many-to-one mapping from $\text{dom}(x_2)$ to $\text{dom}(x_3)$, and R_3 is a one-to-many mapping between $\text{dom}(x_3)$ and $\text{dom}(x_4)$. It is easy to check that this instance has input size $\Theta(N)$ and the output size is OUT . The plan on the left in Figure 1 incurs a cost of $\Theta(N \cdot \text{OUT})$ since $|R_1 \bowtie R_2| = \frac{N \cdot \text{OUT}}{2}$, while the plan on the right incurs a cost of $\Theta(N)$ since $|R_2 \bowtie R_3| = N$ and $|R_1 \times \text{dom}(x_4)| = N$.

Next, we construct a symmetric instance (see the bottom half of Figure 2). Now, these two plans would have the opposite behavior. Finally, we consider the instance by combining these two sub-instances together. On this combined instance, both plans incur a cost of $\Theta(N \cdot \text{OUT})$, since $|R_1 \bowtie R_2| = |R_2 \bowtie R_3| = \Theta(N \cdot \text{OUT})$.

We can apply a similar argument to all other possible query plans for evaluating this instance, and any single query plan has to materialize $\Theta(N \cdot \text{OUT})$ intermediate join results. Hence, Yannakakis algorithm indeed requires $\Theta(N \cdot \text{OUT})$ time on this instance.

Inspired by this instance, we can leverage the power of multiple query plans to overcome the fundamental limitation of the Yannakakis' algorithm. In our example, we want to run the upper part of the input using the TD rooted at (x_4x_5) , and then the lower part using the TD rooted at (x_1x_2) ; this leads to a running time of only $O(N + \text{OUT})$. However, in general, it is not as straightforward to see which TDs should be chosen, how the data must be partitioned, and which part we should send to each TD. In the next section, we will show our novel strategy for the class of chain queries before we generalize it to all acyclic CQs.

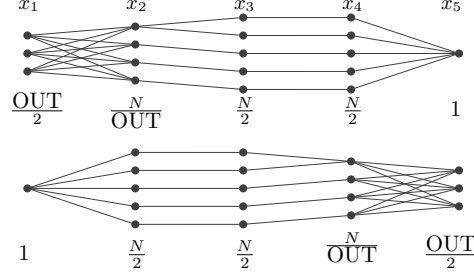


Figure 2: A database instance for 4-chain CQ.

Algorithm 1: CHAINCQ($Q_{\text{chain}}, \mathcal{D}$)

```

1 foreach  $i \in [k-1]$  do
2   if  $i = 1$  then  $T_1(x_1, x_2) \leftarrow R_1$ ;
3   else  $T_i(x_1, x_{i+1}) \leftarrow S_{i-1} \wedge R_i$ ;
4    $x_{i+1}^{\text{heavy}}, x_{i+1}^{\text{light}} \leftarrow$ 
      $\left\{ a \in \text{dom}(x_{i+1}) : |\sigma_{x_{i+1}=a} T_i| > \sqrt{\text{OUT}} \right\},$ 
      $\left\{ a \in \text{dom}(x_{i+1}) : 1 \leq |\sigma_{x_{i+1}=a} T_i| \leq \sqrt{\text{OUT}} \right\}$ ;
5    $R_i^{\text{heavy}}, R_i^{\text{light}} \leftarrow R_i \bowtie x_{i+1}^{\text{heavy}}, R_i \bowtie x_{i+1}^{\text{light}}$ ;
6    $S_i(x_1, x_{i+1}) \leftarrow T_i \bowtie x_{i+1}^{\text{light}}$ ;
7 foreach  $i \in [k-1]$  do
8    $Q_i(x_1, x_{k+1}) \leftarrow \left( \bigwedge_{j \in [i-1]} R_j^{\text{light}} \right) \wedge R_i^{\text{heavy}} \wedge \left( \bigwedge_{j=i+1}^k R_j \right)$ ;
9  $Q_*(x_1, x_{k+1}) \leftarrow \left( \bigwedge_{j \in [k-1]} R_j^{\text{light}} \right) \wedge R_k$ ;
10 return  $Q_1 \cup Q_2 \cup \dots \cup Q_{k-1} \cup Q_*$ ;
```

3.3 New Algorithm

Given the wide variety of TDs for a chain query, one natural question arises: which query plans should we select? For chain queries, we only consider two query plans that correspond to two width-1 TDs: (a) one is rooted at (x_1x_2) ; (b) one is rooted at (x_kx_{k+1}) . See Figure 1 for Q_4 . Behind our hybrid strategy, the idea is to partition the input instance into a set of sub-instances and then choose one of two plans for each sub-instance. For example, if the active domain of x_{k+1} is small, we choose the one rooted at (x_1x_2) ; and if the domain of x_1 is small, we choose the TD rooted at (x_kx_{k+1}) .

As described in Algorithm 1, our new algorithm consists of two stages. In **Stage I**, we partition the input instance. In **Stage II**, we choose different query plans for each sub-instance, apply the Yannakakis algorithm, and union the results of all subqueries. See an example in Figure 3. We assume a constant-approximation of OUT is known; this assumption can be removed without increasing the complexity asymptotically [12]. In the following, we use OUT to denote a constant-approximation of the output size.

Stage I: Partition. For each value $a \in \text{dom}(x_2)$, we define its *degree* as the number of tuples from R_1 that have value a in x_2 , i.e., $\Delta(a) = |\pi_{x_1} \sigma_{x_2=a} R_1|$. A value $a \in \text{dom}(x_2)$ is *heavy* if $\Delta(a) > \sqrt{\text{OUT}}$, and *light* otherwise. Let $A_2^{\text{heavy}}, A_2^{\text{light}}$ be the set of heavy and light values in x_2 . Let $R_1^{\text{heavy}} = R_1 \bowtie A_2^{\text{heavy}}$ and $R_1^{\text{light}} = R_1 \bowtie A_2^{\text{light}}$ be the set of heavy and light tuples in R_1 respectively.

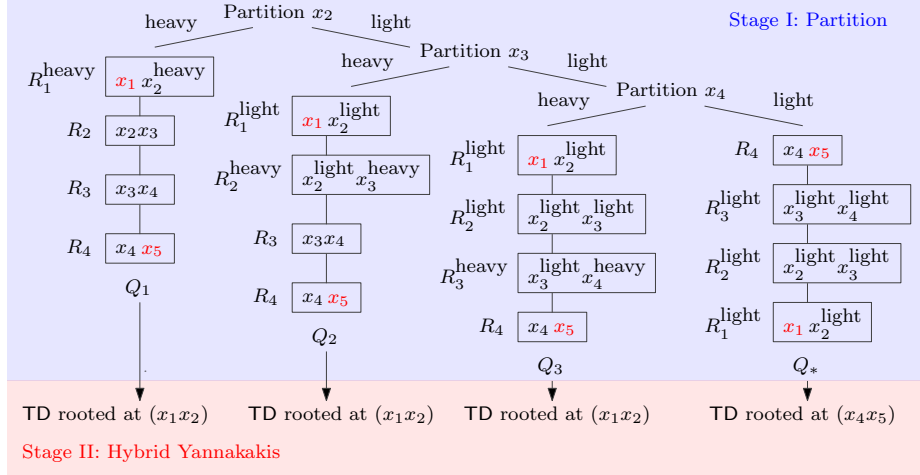


Figure 3: An illustration of Algorithm 1 for the 4-chain query.

We partition relations by ordering $R_2, R_3, \dots, R_k$. Suppose we are done with $R_1, R_2, \dots, R_{i-1}$. We next partition values in $\text{dom}(x_{i+1})$ that can be joined with any value in $\text{dom}(x_1)$ via $R_1^{\text{light}}, R_2^{\text{light}}, \dots, R_{i-1}^{\text{light}}$. Let

$$\Delta(a) = \left| \pi_{x_1} \left\{ \left(\bowtie_{j \in [i-1]} R_j^{\text{light}} \right) \bowtie (\sigma_{x_{i+1}=a} R_i) \right\} \right|$$

be the *degree* of each value $a \in \text{dom}(x_{i+1})$. A value $a \in \text{dom}(x_{i+1})$ is *heavy* if $\Delta(a) > \sqrt{\text{OUT}}$, and *light* otherwise. Let $x_{i+1}^{\text{heavy}}, x_{i+1}^{\text{light}}$ be the set of heavy, light values in x_i . Then, $R_i^{\text{heavy}} = R_i \bowtie x_{i+1}^{\text{heavy}}$ and $R_i^{\text{light}} = R_i \bowtie x_{i+1}^{\text{light}}$. Note that some values in x_{i+1} may be undefined, as well as some tuples in R_i .

Instead of computing $\Delta(\cdot)$ directly, we introduce the following intermediate relations:

$$T_i(x_1, x_{i+1}) = \pi_{x_1, x_{i+1}} \left(\bowtie_{j \in [i-1]} R_j^{\text{light}} \right) \bowtie R_i,$$

$$S_i(x_1, x_{i+1}) = \pi_{x_1, x_{i+1}} \left(\bowtie_{j \in [i]} R_j^{\text{light}} \right)$$

and recursively compute them by ($T_1 = R_1$ in line 2):

$$T_i = \pi_{x_1, x_{i+1}} S_i \bowtie R_i; \quad (\text{line 3})$$

$$S_i = \pi_{x_1, x_{i+1}} T_i \bowtie x_{i+1}^{\text{light}}; \quad (\text{line 6})$$

Once we have computed T_i , we can identify the heavy and light values in x_{i+1} , i.e., x_{i+1}^{heavy} and x_{i+1}^{light} (line 4). We can use $x_{i+1}^{\text{heavy}}, x_{i+1}^{\text{light}}$ to partition R_i into $R_i^{\text{heavy}}, R_i^{\text{light}}$ (line 5). Then, S_i can be computed based on T_i and x_{i+1}^{light} . Furthermore, T_{i+1} can be computed based on S_i and R_{i+1} . We partition Q into k sub-instances:

$$Q_i(x_1, x_{k+1}) \leftarrow \left(\bigwedge_{j \in [i-1]} R_j^{\text{light}} \right) \wedge R_i^{\text{heavy}} \wedge \left(\bigwedge_{j=i+1}^k R_j \right)$$

for $i \in [k-1]$ and $Q_*(x_1, x_{k+1}) \leftarrow \left(\bigwedge_{j \in [k-1]} R_j^{\text{light}} \right) \wedge R_k$.

Stage II: Hybrid Yannakakis. We invoke the Yannakakis algorithm to compute Q_i for each $i \in [k-1]$ using the TD rooted at (x_1, x_2) , and Q_* using the TD rooted at (x_4, x_5) . Finally, we union the results of all subqueries.

Analysis. In **Stage I**, consider any $i \in [k] - \{1\}$. As there are N tuples in R_i , and each of them can be joined

with at most $\sqrt{\text{OUT}}$ tuples in S_{i-1} , T_i can be computed in $O(N \cdot \sqrt{\text{OUT}})$ time. Also, $|T_i| = O(N \cdot \sqrt{\text{OUT}})$. The cost of computing S_i is $O(|T_i|) = O(N \cdot \sqrt{\text{OUT}})$. In **Stage II**, for Q_i , each active value in x_{k+1} can be joined with at least $\sqrt{\text{OUT}}$ values in x_1 , implied by x_{i+1}^{heavy} . As there are OUT results in total, the active domain size of x_{k+1} is $O(\sqrt{\text{OUT}})$. Hence, the number of intermediate join results materialized for each node is at most $O(N \cdot \sqrt{\text{OUT}})$. For Q_* , the total number of intermediate join results is $O(N \cdot \sqrt{\text{OUT}})$, as there are N tuples in R_k and each of them can be joined with at most $\sqrt{\text{OUT}}$ tuples in S_{k-1} . Hence, this step takes $O(N \cdot \sqrt{\text{OUT}})$ time. Finally, as each subquery produces at most OUT results, and there are $O(1)$ subqueries, the aggregation step takes $O(\text{OUT})$ time. Putting everything together, we obtain:

Proposition 1. For Q_{chain} , and an arbitrary instance $\mathcal{D}$ of input size N and output size OUT , the query result $Q(\mathcal{D})$ can be computed in $O(N + N \cdot \sqrt{\text{OUT}})$ time.

4. MAIN RESULTS

In this section, we present our main result. We start with the formal definition of free-connex fractional hypertree width, state the main result, and then show the general algorithm for acyclic CQs. Finally, we extend our result to cyclic CQs and CQs with aggregation. Due to the page limit, all missing details can be found at [12].

4.1 Main Theorem

One of our main results is to identify that free-connex fractional hypertree width is the correct quantity for capturing the output-optimality of acyclic CQs:

Definition 1 (Free-connex Fractional Hypertree Width). For a CQ $Q = (\mathcal{H}, \mathcal{F})$, let $\text{FTD}(Q)$ be the set of all free-connex TDs for Q . The free-connex fractional hypertree

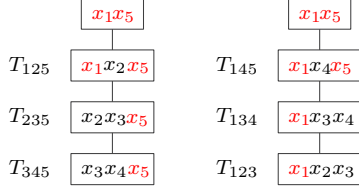


Figure 4: Two free-connex tree decompositions of the 4-chain CQ Q_{chain4} . $T_{\chi(u)}$ shows the attributes in the node of the tree decomposition.

width of Q , denoted as $\text{fn-fhtw}(Q)$, is defined as:

$$\text{fn-fhtw}(Q) := \min_{(\mathcal{T}, \chi) \in \text{FTD}(Q)} \text{width}(\mathcal{T}, \chi)$$

In other words, it is the minimum width of all possible free-connex TDs. Contrast this with the traditional notion of fractional hypertree width, which instead considers all possible TDs. By definition, $\text{fn-fhtw}(Q) \geq 1$ for all CQs and $\text{fn-fhtw}(Q) = 1$ for free-connex CQs.

Example 2. Figure 4 shows two free-connex tree decompositions for Q_4 . Each of the two TDs has width 2, and one can show that $\text{fn-fhtw}(Q_{\text{chain4}}) = 2$. In fact, for the general chain query Q_{chain} , $\text{fn-fhtw}(Q_{\text{chain}}) = 2$.

We are now ready to state our main theorem.

Theorem 1. For any acyclic CQ Q and a database $\mathcal{D}$ with input size N and output size OUT , there is an algorithm that can compute $Q(\mathcal{D})$ in time

$$O\left(N + \text{OUT} + N \cdot \text{OUT}^{1 - \frac{1}{\text{fn-fhtw}(Q)}}\right).$$

4.2 Structural Analysis

To achieve Theorem 1, we point out a helper lemma that can significantly simplify the query structure.

Definition 2 (Separated CQ). A CQ $Q = (\mathcal{H}, \mathcal{F})$ is separated if there is a one-to-one correspondence between the set of output attributes and the set of attributes that only appear in one relation, and for each $e \in \mathcal{E}$ that contain some output attributes, there exists some relation $e' \in \mathcal{E} - \{e\}$ that contains all non-output attributes in e , i.e., $\mathbf{x}_e - \mathbf{x}_{\mathcal{F}} \subseteq \mathbf{x}_{e'}$.

Lemma 1. An acyclic CQ Q with an instance $\mathcal{D}$ of input size N can be transformed into a set of k separated acyclic CQs $Q_1, Q_2, \dots, Q_k$ and k sub-instances $\mathcal{D}_1, \mathcal{D}_2, \dots, \mathcal{D}_k$ each of input size $O(N)$, such that $Q(\mathcal{D}) = \bigwedge_{i \in [k]} Q(R_i)$, $|Q_i(\mathcal{D}_i)| \leq |Q(\mathcal{D})|$ for any $i \in [k]$, and $\text{fn-fhtw}(Q) = \max_{i \in [k]} \text{fn-fhtw}(Q_i)$, in $O(N)$ time.

With Lemma 1, we can focus on evaluating each separated CQ first, and then combining their query result via join, since $Q(\mathcal{D}) = \bigwedge_{i \in [k]} Q(R_i)$, which is also the final query result. This join step will only take $O(N + \text{OUT})$ time. In Section 4.3, we present an output-optimal algorithm for separated acyclic CQs.

We start with a nice structural property of separated acyclic CQs regarding fn-fhtw . In a TD $(\mathcal{T}, \chi)$, for a node e , let $\mathcal{N}_e$ denote the set of nodes incident

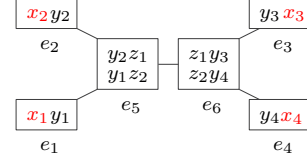


Figure 5: A separated TD $(\mathcal{T}, \chi)$ for $Q_{\text{separated}}$. The set of output attributes, or the set of attributes appearing in only one relation, is $\{x_1, x_2, x_3, x_4\}$ (highlighted in red). The set of boundary nodes, or the set of relations containing output attributes, is $\{e_1, e_2, e_3, e_4\}$.

to it. Node e is in the *boundary* of $\mathcal{T}$ if $|\mathcal{N}_e| = 1$. Throughout this subsection, we will use a more complicated CQ $Q_{\text{separated}}(x_1, x_2, x_3, x_4) \leftarrow R_1(x_1, y_1) \wedge R_2(x_2, y_2) \wedge R_3(x_3, y_3) \wedge R_4(x_4, y_4) \wedge R_5(y_1, y_2, z_1, z_2) \wedge R_6(y_3, y_4, z_3, z_4)$ as the running example.

Lemma 2. Any separated acyclic CQ $Q = (\mathcal{H}, \mathcal{F})$ has a width-1 TD $(\mathcal{T}, \chi)$ such that there is a one-to-one correspondence between the set of relations containing output attribute(s) of Q and the boundary of $\mathcal{T}$. $(\mathcal{T}, \chi)$ is called a separated TD for Q .

For a pair of incident nodes e_1, e_2 in $\mathcal{T}$, we use $\{e_1, e_2\}$ to denote the undirected edge between them, use (e_1, e_2) (resp. (e_2, e_1)) to denote the directed edge from e_1 to e_2 (resp. from e_2 to e_1). Removing edge $\{e_1, e_2\}$ separates $\mathcal{T}$ into two connected subtrees $\mathcal{T}_{e_1, e_2}$ and $\mathcal{T}_{e_2, e_1}$, containing e_1 and e_2 separately. Let $\mathcal{L}_{e_1, e_2}$ be the set of boundary nodes in $\mathcal{T}_{e_1, e_2}$. We define $\phi_{e_1, e_2} = \frac{|\mathcal{L}_{e_1, e_2}|}{\text{fn-fhtw}(Q)}$ as the *fraction* of boundary nodes in $\mathcal{T}_{e_1, e_2}$. Each edge (e_1, e_2) derives a sub-query as:

$$Q_{e_1, e_2}(\mathbf{x}_{\mathcal{F}} \cup (\mathbf{x}_{e_1} \cap \mathbf{x}_{e_2})) \leftarrow \bigwedge_{u \in \text{nodes}(\mathcal{T}_{e_1, e_2})} R_u$$

i.e., projecting away all non-output attributes except the join attributes between e_1 and e_2 .

Example 3. Continue with the separated TD for $Q_{\text{separated}}$ in Figure 5. Removing edge $\{e_5, e_6\}$ leads to two sub-trees: $\mathcal{T}_{e_5, e_6}$ contains nodes e_1, e_2, e_5 and $\mathcal{T}_{e_6, e_5}$ contains nodes e_3, e_4, e_6 . Edge (e_5, e_6) derives a CQ $Q_{e_5, e_6}(z_1, z_2, x_1, x_2) \leftarrow R_1(x_1, y_1) \wedge R_2(x_2, y_2) \wedge R_5(y_1, y_2, z_1, z_2)$. And, $\phi(e_5, e_6) = \phi(e_6, e_5) = \frac{1}{2}$.

Definition 3 (Edge Label). For a separated acyclic query Q and a separated TD $(\mathcal{T}, \chi)$, an instance $\mathcal{D}$ and parameter OUT , an edge (e_1, e_2) in $\mathcal{T}$ is

- *large* if $|Q_{e_1, e_2}(\mathcal{D}) \times t| > \text{OUT}^{\phi_{e_1, e_2}}$ holds for every tuple $t \in \pi_{\mathbf{x}_{e_1} \cap \mathbf{x}_{e_2}} R_{e_1}$; and
- *small* if $|Q_{e_1, e_2}(\mathcal{D}) \times t| \leq \text{OUT}^{\phi_{e_1, e_2}}$ holds for every tuple $t \in \pi_{\mathbf{x}_{e_1} \cap \mathbf{x}_{e_2}} R_{e_1}$; and
- *unlabeled* otherwise.

Furthermore, we distinguish a small edge (e_1, e_2) as *limited* if $|\pi_{\mathbf{x}_{\mathcal{F}}} Q_{e_1, e_2}(\mathcal{D})| \leq \text{OUT}^{\phi_{e_1, e_2}}$.

Algorithm 2: HYBRIDYANNAKAKIS($Q, \mathcal{D}$)

```

1  $(\mathcal{T}, \chi) \leftarrow$  a separated TD of  $Q$  with all edges
   non-labeled;
2  $\mathcal{P} \leftarrow \{(\mathcal{T}, \mathcal{D})\}$ ;
3 while  $\mathcal{P} \neq \emptyset$  do
4    $(\mathcal{T}', \mathcal{D}') \leftarrow$  an arbitrary pair in  $\mathcal{P}$ ;
5   while  $\exists$  non-labeled  $(e_1, e_2)$  in  $\mathcal{T}'$  such that
      $(e_3, e_1)$  is limited for each  $e_3 \in \mathcal{N}_{e_1} - \{e_2\}$  do
6     Label edge  $(e_1, e_2)$  as limited;
7   if  $(\mathcal{T}', \mathcal{D}')$  meets Lemma 5 then
8     Computing  $Q(\mathcal{D}')$ ;
9      $S \leftarrow S \cup Q(\mathcal{D}')$ ;
10  else
11     $(e_1, e_2) \leftarrow$  a non-labeled edge in  $\mathcal{T}'$  such that
       $(e, e_1)$  is small for any  $e \in \mathcal{N}_{e_1} - \{e_2\}$ ;
12    Compute  $Q_{e_1, e_2}(\mathcal{D}')$ ;
13     $\mathcal{H} \leftarrow$  the set of tuples  $t$  in  $\text{dom}(\mathbf{x}_{e_1} \cap \mathbf{x}_{e_2})$ 
      such that
14     $|\sigma_{\mathbf{x}_{e_1} \cap \mathbf{x}_{e_2} = t} Q_{e_1, e_2}(\mathcal{D}')| > \text{OUT}^{\phi_{e_1, e_2}}$ ;
       $\mathcal{T}'_1 \leftarrow \mathcal{T}'$  with  $(e_1, e_2)$ ,  $(e_2, e_1)$  labeled as
      large, limited separately;
15     $\mathcal{T}'_2 \leftarrow \mathcal{T}'$  with  $(e_1, e_2)$  labeled as small;
16    Add  $(\mathcal{T}'_1, \mathcal{D}' - \{R_{e_1}\} \cup \{R_{e_1} \ltimes \mathcal{H}\})$  to  $\mathcal{P}$ ;
17    Add  $(\mathcal{T}'_2, \mathcal{D}' - \{R_{e_1}\} \cup \{R_{e_1} \triangleright \mathcal{H}\})$  to  $\mathcal{P}$ ;
18    Remove  $(\mathcal{T}', \mathcal{D}')$  from  $\mathcal{P}$ ;
19 return  $S$ ;

```

Lemma 3 (Limited-Implied-Limited). *For any node e_1 , if there exists a node $e_2 \in \mathcal{N}_{e_1}$ such that edge (e_3, e_1) is limited for every node $e_3 \in \mathcal{N}_{e_1} - \{e_2\}$, then edge (e_1, e_2) must be limited.*

Lemma 4 (Large-Reverse-Limited). *If edge (e_1, e_2) is large, then edge (e_2, e_1) must be limited.*

4.3 Algorithm

Base Case. We start by characterizing an *optimal condition* on the input instances for which the runtime of the Yannakakis algorithm can be bounded as stated in Theorem 1. We call such an instance a base case.

Lemma 5 (Base Case). *For a separated acyclic CQ $Q = (\mathcal{H}, \mathcal{F})$ with a separated TD $(\mathcal{T}, \chi)$, and an instance $\mathcal{D}$ of input size N and output size OUT , if there is a boundary node e of $\mathcal{T}$ such that the only edge $(*, e)$ is small, then Yannakakis algorithm can compute $Q(\mathcal{D})$ in $O(N + \text{OUT} + N \cdot \text{OUT}^{1 - \frac{1}{\text{fn-htw}(Q)}})$ time.*

Partition into Base Cases. The Yannakakis algorithm falls short of optimality when input instances fail to meet the required condition in Lemma 5. Even worse, it is unknown how to efficiently *decide* the label of each edge because the definitions rely on sub-queries that are expensive to compute. Therefore, the core technical challenge is defining an efficient ordering for edge labeling and isolating sub-instances once they fulfill the criteria of Lemma 5.

As described in Algorithm 2, let $(\mathcal{T}, \chi)$ be a separated TD for Q (line 1), with all edges unlabeled initially. We put $(\mathcal{T}, \mathcal{D})$ into a candidate set $\mathcal{P}$ of instances to be partitioned (line 2). In general, consider an arbitrary pair $(\mathcal{T}', \mathcal{D}') \in \mathcal{P}$. From Lemma 3, we apply *limited-imply-limited* rule to infer edge labels (lines

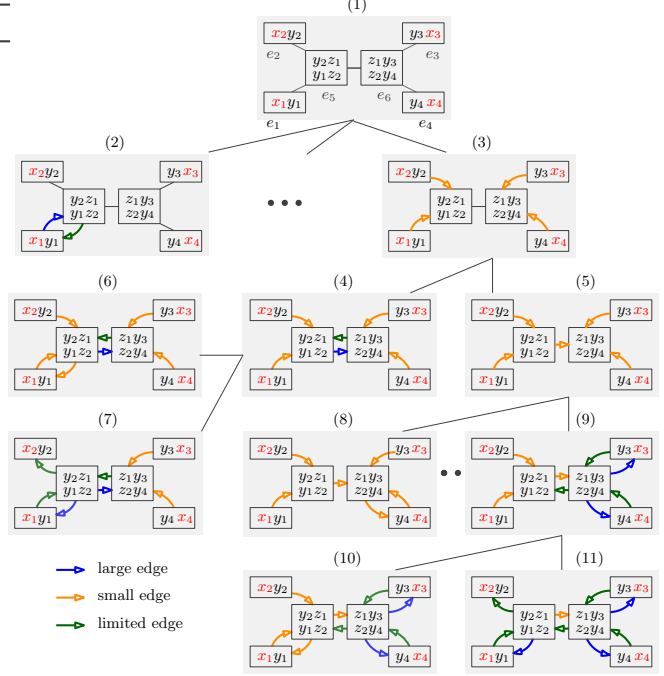


Figure 6: An illustration of the partition procedure on the query in Figure 5.

5-6). If it meets the optimal condition, we compute the result immediately (lines 7-9). Otherwise, we further partition it (lines 9-17). More specially, we pick a non-labeled edge (e_1, e_2) such that every other incoming edge to e_1 is small, i.e., edge (e_3, e_1) is small for each node $e_3 \in \mathcal{N}_{e_1} - \{e_2\}$ (line 11). We compute $Q_{e_1, e_2}(\mathcal{D}')$ using the Yannakakis algorithm along $\mathcal{T}_{e_1, e_2}$ rooted at e_1 (line 12). A tuple $t \in \text{dom}(\mathbf{x}_{e_1} \cap \mathbf{x}_{e_2})$ is *heavy* if $|Q_{e_1, e_2}(\mathcal{D}') \ltimes t| > \text{OUT}^{\phi_{e_1, e_2}}$, and *light* otherwise. Now, we construct two sub-instances for $\mathcal{D}'$, which contain heavy and light tuples in R_{e_1} separately (line 16-17), and two copies of $\mathcal{T}'$ in which edge (e_1, e_2) is further labeled as large and small separately (line 14-15). By Lemma 4, we can apply *large-reverse-limited* rule to infer (e_2, e_1) as limited when (e_1, e_2) is labeled as large (line 14). We add these two sub-instances into $\mathcal{P}$ (line 16-17). In either case, we will remove $(\mathcal{T}', \mathcal{D}')$ from $\mathcal{P}$ (line 18). We continue applying this procedure to every remaining pair in $\mathcal{P}$ until $\mathcal{P}$ becomes empty (line 3).

Example 4. We continue the example in Figure 5. Initially, all edges are unlabeled in (1). We start with applying line 13 to label edge (e_1, e_5) since $\mathcal{N}_{e_1} - \{e_5\} = \emptyset$. In (2), the instance with large edge (e_1, e_5) and limited edge (e_5, e_1) already meets the optimal condition. The remaining instance has a small edge (e_5, e_1) . We can apply a similar argument to edges (e_2, e_5) , (e_3, e_6) and (e_4, e_6) . In (3), we are left with the remaining instance with small edges (e_1, e_5) , (e_2, e_5) , (e_3, e_6) and (e_4, e_6) . Then, we can apply line 13 to label edge (e_5, e_6) .

In (4), for instance, with large edge (e_5, e_6) and limited edge (e_6, e_5) , we can apply line 13 to partition both edges (e_5, e_1) and (e_5, e_2) . Suppose we partition

edge (e_5, e_1) wlog. In (6), the instance with small edge (e_5, e_1) already meets the optimal condition. In (7), we are left with an instance with a large edge (e_5, e_1) and a limited edge (e_1, e_5) . We can apply the limited-imply-limited rule to infer edge (e_5, e_2) as limited. This instance also meets the optimal condition.

In (5), for the other instance with small edge (e_5, e_6) , we can apply line 13 to label both edges (e_6, e_4) and (e_6, e_5) . Suppose we label edge (e_6, e_4) wlog. In (8), the instance with small edge (e_6, e_4) meets the optimal condition. We can apply a similar argument to edge (e_6, e_3) . In (9), the remaining instance has large edges (e_6, e_3) and (e_6, e_4) , as well as limited edges (e_3, e_6) and (e_4, e_6) . We can apply the limited-imply-limited rule to infer edge (e_6, e_5) as limited. Now, we can apply line 13 to label edges (e_5, e_1) and (e_5, e_2) . Suppose we label edge (e_5, e_1) . In (10), the instance with small edge (e_5, e_1) already meets the optimal condition. In (11), we are left with the instance with large edge (e_3, e_1) and limited edge (e_3, e_2) . Then, we can apply the limited-imply-limited rule to infer edge (e_3, e_2) as limited. This instance also meets the optimal condition. Now, $\mathcal{P} = \emptyset$, and we are done with the partition procedure.

4.4 Extensions

Below, we discuss two nice extensions of our output-optimal algorithm for acyclic CQs:

Cyclic CQs. Cyclic queries are usually tackled with tree decomposition techniques. Given a tree decomposition of the cyclic CQ, we can first evaluate the derived query for each bag, materialize the query results as a relation, and then apply our new output-optimal algorithm to the transformed acyclic CQ. [3, 16] showed an algorithm that can decompose any cyclic CQ with a database instance $\mathcal{D}$ of input size N into a set of tree decompositions with sub-instances, such that the input size of each sub-instance is bounded by at most $O(N^{\text{subw}(Q)})$, where $\text{subw}(Q)$ is the sub-modular width of Q [17].

Theorem 2. *For any CQ Q and a database $\mathcal{D}$, there is an algorithm that can compute $Q(\mathcal{D})$ in*

$$O\left(N + \text{OUT} + N^{\text{subw}(Q)} \cdot \text{OUT}^{1 - \frac{1}{\max_i \text{fn-fhtw}(Q_i)}}\right)$$

time, where $\{Q_i\}_i$ is the set of acyclic queries that correspond to TDs of Q .

CQs with Aggregation. Let $(\mathbf{D}, \oplus, \otimes, \mathbf{0}, \mathbf{1})$ be a commutative semiring.¹ Some commonly used commutative semiring include Boolean semiring $\mathbb{B} = (\{\text{false}, \text{true}\}, \vee, \wedge, \text{false}, \text{true})$, counting semiring $\mathbb{C} = (\mathbb{N}, +, \cdot, 0, 1)$, and tropical semiring $\text{Trop}^+ = (\mathbb{R}_+ \cup \{\infty\}, \min, +, \infty, 0)$. Abo Khamis et al. [2] introduced the functional aggregate queries (FAQ) that express CQs with aggregation via semiring annotations.

¹A commutative semiring $(\mathbf{D}, \oplus, \otimes, \mathbf{0}, \mathbf{1})$ satisfies the following properties: (1) $(\mathbf{D}, \oplus, \mathbf{0})$ is a commutative monoid with additive identity $\mathbf{0}$ (i.e., $\oplus$ is associative and commutative, and $a \oplus \mathbf{0} = a$ for all $a \in \mathbf{D}$); (2) $(\mathbf{D}, \otimes, \mathbf{1})$ is a (commutative) monoid with multiplicative identity $\mathbf{1}$ for $\otimes$; (3) $\otimes$ distributes over $\oplus$, i.e., $a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$ for $a, b, c \in \mathbf{D}$; and (4) $a \otimes \mathbf{0} = \mathbf{0}$ for all $a \in \mathbf{D}$.

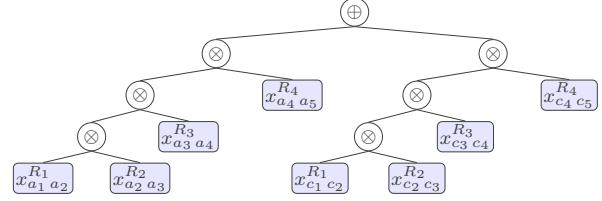


Figure 7: Arithmetic circuit for the 4-chain full CQ. Leaves are monomials representing the tuples; internal nodes are $\oplus$ and $\otimes$.

Let $Q = (\mathcal{H}, \mathcal{F})$ be a CQ where $\mathcal{H} = (\mathcal{V}, \mathcal{E})$. A FAQ φ over a CQ Q and a commutative semiring $(\mathbf{D}, \oplus, \otimes, \mathbf{0}, \mathbf{1})$ is defined as follows:

$$\varphi(\mathbf{x}_{\mathcal{F}}) \leftarrow \bigoplus_{\mathcal{V} \setminus \mathbf{x}_{\mathcal{F}}} \bigotimes_{e \in \mathcal{E}} R_e(\mathbf{x}_e),$$

Let Q_{full} be the full version of Q : $Q_{\text{full}}(\mathcal{V}) \leftarrow \bigwedge_{e \in \mathcal{E}} R_e(\mathbf{x}_e)$. For a database instance $\mathcal{D}$, each relation $R_e(\mathbf{x}_e)$ is enriched into a σ -relation, where each tuple is associated with a semi-ring element from $\mathbf{D}$. The query result of ψ on $\mathcal{D}$, denoted as $\psi(\mathcal{D})$ is a σ -relation of schema $\mathbf{x}_{\mathcal{F}}$, where each query result $s_{\mathbf{x}_{\mathcal{F}}} \in Q(\mathcal{D})$ is associated with the following semi-ring element from $\mathbf{D}$:

$$p_{\mathcal{D}[s_{\mathbf{x}_{\mathcal{F}}}] }^{\mathcal{H}} := \bigoplus_{t \in Q_{\text{full}}(\mathcal{D}) : \pi_{\mathbf{x}_{\mathcal{F}}} t = s_{\mathbf{x}_{\mathcal{F}}}} \bigotimes_{J \in \mathcal{E}} x_{(\pi_{\mathbf{x}_J} t)}^e \quad (1)$$

where $x_{(\pi_{\mathbf{x}_e} t)}^e$ is the semi-ring element from $\mathbf{D}$ associated with tuple $(\pi_{\mathbf{x}_e} t) \in R_e$. Our output-sensitive algorithm can be similarly extended to acyclic FAQs with the following result:

Theorem 3. *For an FAQ query ψ over an acyclic CQ Q , a commutative semiring $(\mathbf{D}, \oplus, \otimes, \mathbf{0}, \mathbf{1})$, and a database $\mathcal{D}$, there is an algorithm that can compute the query result in $O\left(N + N \cdot \text{OUT}^{1 - \frac{1}{\text{fn-fhtw}(Q)}} + \text{OUT}\right)$ time.*

5. LOWER BOUND

We now present a matching lower bound by showing the smallest circuit needed for computing the provenance polynomial of all query results over a given semiring. The *sum-product polynomial* for a CQ $Q = (\mathcal{H}, \mathcal{F})$, is parameterized by an underlying semiring, a hypergraph $\mathcal{H}$, an instance $\mathcal{D}$ and a tuple $s_{\mathbf{x}_{\mathcal{F}}} \in Q(\mathcal{D})$, is essentially $p_{\mathcal{D}[s_{\mathbf{x}_{\mathcal{F}}}] }^{\mathcal{H}}$ defined in Equation 1.

Circuits over Semirings. A circuit F over a semiring is a *directed acyclic graph* with input nodes (with fan-in 0) variables in a set S_x containing $x_{(\pi_{\mathbf{x}_e} t)}^e$'s in the right-hand-side of Equation 1 and the constants $\mathbf{0}, \mathbf{1}$. Every other node is labeled by $\oplus$ or $\otimes$ and has fan-in 2; these nodes are called $\oplus$ -gates and $\otimes$ -gates, respectively. An input gate of F is any gate with fan-in 0, and an output gate of F is any gate with fan-out 0. The size of the circuit F , denoted as $|F|$, is the number of gates in F .

Example 5. *Consider the full version of 4-chain CQ: $Q_{\text{full}}(x_1, x_2, x_3, x_4) \leftarrow R_1(x_1, x_2) \wedge R_2(x_2, x_3) \wedge R_3(x_3, x_4) \wedge R_4(x_4, x_5)$ with a database instance $\mathcal{D}$: $R_1 = \{(a_1, a_2), (c_1, c_2)\}$; $R_2 = \{(a_2, a_3), (c_2, c_3)\}$;*

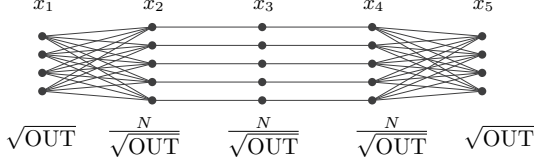


Figure 8: A database instance of 4-chain query.

$R_3 = \{(a_3, a_4), (c_3, c_4)\}$; and $R_4 = \{(a_4, a_5), (c_4, c_5)\}$. Figure 7 shows the circuit that computes the polynomial shown below

$$p_{\mathcal{D}}^{Q_{full}} = x_{a_1 a_2}^{R_1} \otimes x_{a_2 a_3}^{R_2} \otimes x_{a_3 a_4}^{R_3} \otimes x_{a_4 a_5}^{R_4} \\ \oplus x_{c_1 c_2}^{R_1} \otimes x_{c_2 c_3}^{R_2} \otimes x_{c_3 c_4}^{R_3} \otimes x_{c_4 c_5}^{R_4}$$

The circuit has size 7 as there are three $\otimes$ gates in each term and one $\oplus$ gate that aggregates the final result.

Disjunctive datalog Rules. A *disjunctive datalog rule* [3] is an expression

$$P : \bigvee_{B \in \mathcal{B}} T_B(\mathbf{x}_B) \leftarrow \bigwedge_{J \in \mathcal{E}} R_J(\mathbf{x}_J)$$

where $B \subseteq [n]$ and $\mathcal{B} \subseteq 2^{[n]}$, where $2^{[n]}$ is the power set of $[n]$. The body is similar to that of a CQ, while the head is a disjunction of output relations T_B which are called *targets*. Given an instance $\mathcal{D}$, a *model* of a disjunctive datalog rule is a tuple $\mathbf{T} = (T_B)_{B \in \mathcal{B}}$ of relations, denoted as $\mathbf{T} \models P$, such that for any tuple t , if $(\pi_e t) \in R_e$ for all $e \in \mathcal{E}$, then there exists a target $T_B \in \mathbf{T}$ such that $t[B] \in T_B$.

To find the lower bound, we seek to find the size of the smallest circuit that contains one output gate, each computing $p_{\mathcal{D}}^{\mathcal{H}}[s_U]$ for all $s_U \in Q(\mathcal{D})$, for a semiring $\mathbb{S}$. We begin by stating the matching lower bounds for acyclic queries under the output-sensitive setting.

Theorem 4. *For any acyclic but non-free-connex CQ $Q = (\mathcal{H}, \mathcal{F})$, and parameters $N, \text{OUT} \in \mathbb{N}$ with $\text{OUT} \leq N^{\rho^*}$, there is an instance $\mathcal{D}$ of input size $\Theta(N)$ and output size $\Theta(\text{OUT})$ such that the size of the smallest circuit that computes $p_{\mathcal{D}}^{\mathcal{H}}[s_{\mathcal{F}}]$ for all query result $s_{\mathcal{F}} \in Q(\mathcal{D})$ is $\Omega\left(N \cdot \text{OUT}^{1 - \frac{1}{\text{fr-fnw}(Q)}}\right)$, which holds for each semiring in $\{\text{Trop}^+, \mathbb{C}, \mathbb{B}_{\text{lin}}\}$.*

This lower bound is *unconditional*, i.e., it does not depend on any conjecture. The key technique is to consider the hypergraph with an extra hyperedge $\{\mathbf{x}_{\mathcal{F}}\}$ added, i.e., $Q' = (\mathcal{H}, \mathcal{F} \cup \{\mathbf{x}_{\mathcal{F}}\})$, and then consider the size of the largest bag that can be formed across all disjunctive datalog rules that can be formed for any given set of tree decompositions. We present the intuition of the lower bound result via the following example.

Example 6. *We construct a database instance for 4-chain CQ as shown in Figure 8. Attributes x_1, x_2, x_3, x_4 and x_5 have domain sizes $\sqrt{\text{OUT}}, \frac{N}{\sqrt{\text{OUT}}}, \frac{N}{\sqrt{\text{OUT}}}, \frac{N}{\sqrt{\text{OUT}}}, \sqrt{\text{OUT}}$. R_1 is a Cartesian product between x_1 and x_2 , R_2 is a many-to-one mapping from x_2 to x_3 , R_3 is a one-to-many mapping from x_3 to x_4 and R_4 is a Cartesian product between x_4 and x_5 . It can be easily checked*

that the instance has input size $\Theta(N)$ and the output size is $\Theta(\text{OUT})$. We add another relation R_{OUT} containing $\{x_1, x_5\}$ to the body of the query, making it cyclic.

Consider the two free-connex tree decompositions as shown in Figure 4 and the disjunctive rule below:

$$T_{125} \vee T_{345} \leftarrow R_1 \wedge R_2 \wedge R_3 \wedge R_4 \wedge R_{\text{OUT}}$$

Then, the size of the bag T_{125} is at least $O(N \cdot \sqrt{\text{OUT}})$ since every tuple in relation R_1 has a join path to every attribute value of x_5 .

Any free-connex tree decomposition will contain a bag that contains variables x_1, x_5 along with either x_2 or x_4 . Thus, the worst-case size of such a bag is $\Theta(N \cdot \sqrt{\text{OUT}})$.

6. CONCLUSION

In this paper, we establish matching lower and upper bounds for evaluating general acyclic CQs in an output-optimal manner, characterized by the free-connex fractional hypertree width of the queries. This result generalizes and improves upon all previously known upper and lower bounds. As a by-product, it also implies new output-sensitive algorithms for cyclic CQs as well as CQs with aggregation over commutative semirings.

Our findings open several promising avenues for future investigation:

- *Output Optimality for Cyclic Queries:* The pursuit of output-optimal algorithms for cyclic CQs remains a significant open challenge. Specifically, it is yet to be determined if the free-connex fractional hypertree width serves as the definitive quantity to characterize the output optimality in these contexts.
- *Practical Systems Integration:* Beyond theoretical bounds, there is a clear opportunity to integrate this hybrid Yannakakis framework into practical data systems. This would build upon a growing body of implementation-focused research [25, 22, 23, 7] of the Yannakakis algorithm, bridging the gap between database theory and practical query processing.
- *Non-equi-join Queries:* It is an open problem to design output-sensitive algorithms for set similarity [9] and set containment joins [10]. We posit that some of our techniques can be applied to those queries to obtain novel algorithms with non-trivial guarantees.
- *Fast Matrix Multiplication:* A particularly compelling direction involves incorporating fast matrix multiplication to speed up combinatorial algorithms. Recent breakthroughs [8, 13, 11, 1] demonstrate that the use of algebraic techniques can effectively bypass traditional barriers and accelerate the evaluation of CQs, but it remains to be investigated for general CQs.

7. REFERENCES

- [1] M. Abo Khamis, X. Hu, and D. Suciu. Fast matrix multiplication meets the submodular width. *Proceedings of the ACM on Management of Data*, 3(2):1–26, 2025.

- [2] M. Abo Khamis, H. Q. Ngo, and A. Rudra. Faq: questions asked frequently. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 13–28, 2016.
- [3] M. Abo Khamis, H. Q. Ngo, and D. Suciu. What do shannon-type inequalities, submodular width, and disjunctive datalog have to do with one another? In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 429–444, 2017.
- [4] A. Atserias, M. Grohe, and D. Marx. Size bounds and query plans for relational joins. *SIAM Journal on Computing*, 42(4):1737–1767, 2013.
- [5] G. Bagan, A. Durand, and E. Grandjean. On acyclic conjunctive queries and constant delay enumeration. In *International Workshop on Computer Science Logic*, pages 208–222. Springer, 2007.
- [6] C. Beeri, R. Fagin, D. Maier, and M. Yannakakis. On the desirability of acyclic database schemes. *Journal of the ACM (JACM)*, 30(3):479–513, 1983.
- [7] L. Bekkers, F. Neven, S. Vansummeren, and Y. R. Wang. Instance-optimal acyclic join processing without regret: Engineering the yannakakis algorithm in column stores. *Proc. VLDB Endow.*, 18(8):2413–2426, Apr. 2025.
- [8] S. Deep, X. Hu, and P. Koutris. Fast join project query evaluation using matrix multiplication. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, SIGMOD '20*, page 1213–1223, New York, NY, USA, 2020. Association for Computing Machinery.
- [9] D. Deng, Y. Tao, and G. Li. Overlap set similarity joins with theoretical guarantees. In *Proceedings of the 2018 International Conference on Management of Data*, pages 905–920, 2018.
- [10] D. Deng, C. Yang, S. Shang, F. Zhu, L. Liu, and L. Shao. Lcjoin: Set containment join via list crosscutting. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 362–373. IEEE, 2019.
- [11] X. Hu. Fast matrix multiplication for query processing. *Proceedings of the ACM on Management of Data*, 2(2):1–25, 2024.
- [12] X. Hu. Output-optimal algorithms for join-aggregate queries. *arXiv preprint arXiv:2406.05536*, 2024.
- [13] Z. Huang and S. Chen. Density-optimized intersection-free mapping and matrix multiplication for join-project operations. *Proceedings of the VLDB Endowment*, 15(10):2244–2256, 2022.
- [14] M. Idris, M. Ugarte, and S. Vansummeren. The dynamic Yannakakis algorithm: Compact and efficient query processing under updates. In *Proc. ACM SIGMOD International Conference on Management of Data*, 2017.
- [15] M. A. Khamis, H. Q. Ngo, and D. Suciu. What do shannon-type inequalities, submodular width, and disjunctive datalog have to do with one another? In *Proc. ACM Symposium on Principles of Database Systems*, 2017.
- [16] M. A. Khamis, H. Q. Ngo, and D. Suciu. Pandaexpress: a simpler and faster panda algorithm. *arXiv preprint arXiv:2512.10217*, 2025.
- [17] D. Marx. Tractable hypergraph properties for constraint satisfaction and conjunctive queries. *Journal of the ACM*, 2013.
- [18] H. Q. Ngo. Worst-case optimal join algorithms: Techniques, results, and open problems. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 111–124. ACM, 2018.
- [19] H. Q. Ngo, C. Ré, and A. Rudra. Skew strikes back: New developments in the theory of join algorithms. *ACM SIGMOD Record*, 42(4):5–16, 2014.
- [20] R. Pagh and M. Stöckel. The input/output complexity of sparse matrix multiplication. In *Proc. European Symposium on Algorithms*, 2014.
- [21] T. L. Veldhuizen. Leapfrog triejoin: A simple, worst-case optimal join algorithm. In *Proc. International Conference on Database Theory*, 2014.
- [22] Q. Wang, B. Chen, B. Dai, K. Yi, F. Li, and L. Lin. Yannakakis+: Practical acyclic query evaluation with theoretical guarantees. *Proceedings of the ACM on Management of Data*, 3(3):1–28, 2025.
- [23] Y. Yang, H. Zhao, X. Yu, and P. Koutris. Predicate transfer: Efficient pre-filtering on multi-join queries. *CIDR*, 2024.
- [24] M. Yannakakis. Algorithms for acyclic database schemes. In *VLDB*, volume 81, pages 82–94, 1981.
- [25] J. Zhao, K. Su, Y. Yang, X. Yu, P. Koutris, and H. Zhang. Debunking the myth of join ordering: Toward robust sql analytics. *Proceedings of the ACM on Management of Data*, 3(3):1–28, 2025.

Technical perspective on ‘Differentially Private Substring and Document Counting’

Grigorios Loukides
King’s College London
London, United Kingdom
grigorios.loukides@kcl.ac.uk

ACM Reference Format:

Grigorios Loukides. 2026. Technical perspective on ‘Differentially Private Substring and Document Counting’. In . ACM, New York, NY, USA, 1 page. <https://doi.org/10.1145/nnnnnnnn.nnnnnnnn>

Differential privacy has become the de facto privacy standard, as it is widely employed by various organizations. However, only a few research works have focused on strings (e.g., [1, 2, 5]). This is surprising, given that strings are fundamental in modeling, for example, genomic sequences, mobility traces, or text logs. One of the key tasks in string database analysis is to count the occurrences of string fragments in the database: for instance, extracting frequent patterns or q -grams, or publishing word frequency statistics. Most existing differentially private algorithms for string mining are largely heuristic and come with little or no worst-case error analysis, which is a major limitation given the fact that strings appear in key domains for decision making.

The paper “Differentially Private Substring and Document Counting” by Bernardini, Bille, Gørtz, and Steiner addresses this limitation by providing the first near-tight accuracy bounds and efficient data structures for the SUBSTRING COUNT and DOCUMENT COUNT problems under differential privacy. The SUBSTRING COUNT problem asks to count the total number of occurrences of a given string P (referred to as *pattern*) in a given collection of strings $\mathcal{D}$ (referred to as database). The DOCUMENT COUNT problem asks to count the total number of strings in $\mathcal{D}$ that contain P .

Both problems could intuitively be solved by building a data structure on $\mathcal{D}$, answering a user query on $\mathcal{D}$, and then protecting the query answer by making it differentially private before returning it to the user. However, if one were to answer many queries, then the privacy loss would grow with the number of queries answered, significantly weakening the offered privacy and making the data structure less useful over time. In contrast, the approach by Bernardini et al. is to design a data structure for these problems, which itself (rather than each query answer) is differentially private and can be queried for an arbitrary number of patterns without any privacy loss. To enforce privacy, this approach hides the presence or absence of any string in $\mathcal{D}$, as it requires the answer to be (almost) the same in any two neighboring databases that differ by a single

string. To preserve data utility, the data structure outputs a noisy (differentially private) count for any query, while minimizing the additive error.

Such a data structure could be applied, for instance, to analyze DNA databases by counting how often some mutation patterns appear in a population (SUBSTRING COUNT queries) without exposing highly identifying data like ethnicity or family relationships; or to find out how many patients followed specific treatment patterns in a corpus of treatment histories (DOCUMENT COUNT queries) while satisfying legal privacy requirements like HIPAA and GDPR.

While the most general data structure for the strictest model of differential privacy proposed by Bernardini et al. is arguably not yet practical because of the high construction time, their data structure to answer queries for fixed-length q -grams under the less strict model of approximate differential privacy can be constructed in almost linear time in the database size, and it is thus of practical relevance.

Such a practical data structure could be successfully applied, for instance, to publish top-trending query q -grams in corpora of search query sequences without leaking users’ private information like political beliefs or health concerns. Another important and impactful application would be in releasing q -gram frequency statistics in natural language collections, a task that is particularly relevant for training LLMs on private texts.

Interestingly, the techniques in Bernardini et al. can be easily adapted to enforce differential privacy or approximate differential privacy on popular counting problems on trees [3, 4].

References

- [1] Huipeng Chen, Changyu Dong, Liyue Fan, Grigorios Loukides, Solon P. Pissis, and Leen Stougie. Differentially private string sanitization for frequency-based mining tasks. In *IEEE International Conference on Data Mining, ICDM 2021, Auckland, New Zealand, December 7–10, 2021*, pages 41–50, 2021.
- [2] R. Chen, B. CM Fung, B. C Desai, and N. M Sossou. Differentially private transit data publication: a case study on the montreal transportation system. In *KDD*, pages 213–221, 2012.
- [3] Badih Ghazi, Pritish Kamath, Ravi Kumar, Pasin Manurangsi, and Kewen Wu. On differentially private counting on trees. In *Proc. 50th ICALP*, volume 261, pages 66:1–66:18, 2023.
- [4] Badih Ghazi, Ravi Kumar, Jelani Nelson, and Pasin Manurangsi. Private counting of distinct and k -occurring items in time windows. In *Proc. 14th ITCS*, pages 55:1–55:24, 2023.
- [5] S. Xu, X. Cheng, S. Su, K. Xiao, and L. Xiong. Differentially private frequent sequence mining. *TKDE*, 28(11):2910–2926, 2016.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

Conference’17, Washington, DC, USA

© 2026 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM

<https://doi.org/10.1145/nnnnnnnn.nnnnnnnn>

A Differentially Private Data Structure for Substring and Document Counting

Giulia Bernardini
University of Milan
Milan, Italy
giulia.bernardini@unimi.it

Inge Li Gørtz
Technical University of
Denmark
Lyngby, Denmark
inge@dtu.dk

Philip Bille
Technical University of
Denmark
Lyngby, Denmark
phbi@dtu.dk

Teresa Anna Steiner
University of Southern
Denmark
Odense, Denmark
steiner@imada.sdu.dk

ABSTRACT

For databases consisting of many text documents, one of the most fundamental data analysis tasks is counting (i) how often a pattern appears as a substring in the database (*substring counting*) and (ii) how many documents in the collection contain the pattern as a substring (*document counting*). If such a database contains sensitive data, it is crucial to protect the privacy of individuals in the database. Differential privacy is the gold standard for privacy in data analysis. It gives rigorous privacy guarantees, but comes at the cost of yielding less accurate results.

In this paper, we study the problem of substring and document counting under differential privacy. We give the first differentially private data structures for these problems and provide bounds on their additive error. For ϵ -differential privacy, we show that the error of our data structure is optimal up to a poly-logarithmic factor in the number of documents and length of the longest document. Our data structures immediately lead to improved algorithms for related problems, such as privately mining frequent substrings and q -grams.

1. INTRODUCTION

Differential privacy [14] is the gold standard for privacy in data analysis. It has been extensively studied in theory and employed on a large scale by companies such as Google, Apple, and Uber, as well as public institutions such as the US Census Bureau. The goal of differentially private data

analysis is to protect the privacy of any *individual contribution*, while giving information about the *general trends* in the database. This is often achieved as follows: Given a database from which we want to extract some information (specified through a *query*), instead of giving the true output to the query, we add random noise to hide the contribution of any single data point. The random noise is scaled such that for any single data point, any given output should be *roughly equally likely* whether the data point is present in the database or not. Specifically, we say that two databases are *neighboring* if they differ in a single data point. An algorithm is (ϵ, δ) -differentially private if the probabilities of any output differ by at most a factor of e^ϵ and an additive δ between any two neighboring databases. If $\delta = 0$, we call the algorithm ϵ -differentially private. The definition of differential privacy allows us to quantify *how private* the output is. Promising differential privacy naturally comes with a loss of accuracy: a completely random output would be perfectly private, but not very useful. The goal is to achieve a good tradeoff between privacy and accuracy.

Many data analysis applications deal with databases of sequential data, which we refer to as *documents* or *strings*: For example, biological sequence data, web browsing statistics, trajectory data, and text protocols for next-word suggestions. Since these documents contain highly confidential information, an important question is whether such collections can be analyzed while preserving differential privacy. For databases consisting of many documents, we are interested in the *patterns* it contains. In particular: For a database that is a collection of documents $\mathcal{D} = S_1, \dots, S_n$, and a query pattern P , how many documents in $\mathcal{D}$ contain P ? This problem is called DOCUMENT COUNT. Another closely related question is: How often does P appear as a substring of a document in $\mathcal{D}$ *in total*? We call this problem SUBSTRING COUNT. The difference between DOCUMENT COUNT and SUBSTRING COUNT is that in the latter, if a pattern occurs multiple times in the same document of $\mathcal{D}$, we count all its occurrences, while in the former, any document in $\mathcal{D}$ can contribute at most one to the count of any pattern. In the non-private setting, these problems have been extensively studied from a data-structure perspective, where one preprocesses the collection $\mathcal{D}$ to answer queries for any

This work is supported by the Danish Research Council grant number DFF-8021-002498 and the Villum Fonden grant number VIL51463.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright 2025 ACM This is a minor revision of the paper “Differentially Private Substring and Document Counting” published in Proc. ACM Manag. Data 3(2) (PODS), Article 95 (2025). DOI: <https://doi.org/10.1145/3725232>. ...\$5.00.

pattern P efficiently.

In this paper, we study the problem of substring and document counting under differential privacy. Specifically, as in the non-private setting, we are interested in designing a data structure for these problems, which can then be queried for an arbitrary number of patterns. For privacy, we want to *keep any document private*, that is, two neighboring databases differ on a single text document. The data structure should output a noisy count for any pattern query, while minimizing the additive error and preserving differential privacy.

Trying to achieve this poses two main challenges. The first challenge is to provide a data structure that can be queried an arbitrary number of times without any privacy loss: Classic differentially private algorithms are designed as one-shot algorithms, that is, we are only allowed to run them once on any given database. Indeed, running several differentially private algorithms on the same database generally results in a higher privacy loss: As an example, consider the case where we want to output how many people in a database satisfy a given property. A differentially private algorithm computes the true output and adds random noise. Now, if we run the same differentially private algorithm on this database many times, then the average of all outputs will converge to the true output plus the mean of the random noise. Thus, at some point, we will be able to give the true answer with high probability, destroying any privacy guarantee. Summarizing, if one were to answer many queries on the same database by just running a new differentially private algorithm each time, then the privacy loss grows with the number of queries.

The second challenge is the high dimensionality of the input data. Recall that our privacy definition requires protecting the influence of any single document. This limits the accuracy we can hope for: As an example, consider the neighboring databases $\mathcal{D}$ and $\mathcal{D}'$, where the length- ℓ string $S_i = aaa \dots a$ of $\mathcal{D}$ is replaced by the length- ℓ string $S'_i = bbb \dots b$ in $\mathcal{D}'$. The substring count of query pattern $P = a$ differs by ℓ between S_i and S'_i . Thus, intuitively, any algorithm computing the count of P which does not significantly distinguish between $\mathcal{D}$ and $\mathcal{D}'$ must have an error of roughly ℓ . Note that this lower bound holds for the problem of computing the count of a single (and very short) pattern! The high dimensionality also introduces another severe challenge: Note that $P = a$ may not even appear in $\mathcal{D}'$ at all – yet it must have a similar probability of producing a positive count as it does in $\mathcal{D}$. This means that assuming our input data consists of strings of a (known) maximum length ℓ , then in order to satisfy ϵ -differential privacy, *any* string of length at most ℓ must have a positive count with some non-zero probability, not only those which actually appear as substrings in $\mathcal{D}$. On the other hand, we would like to efficiently construct a not-too-large data structure, which prohibits explicitly computing a noisy count for all possible patterns.

We tackle the first challenge by giving a *differentially private construction algorithm* for our data structure; since differential privacy is robust to post-processing, this data structure can then be queried ad-libitum without further privacy loss. Thus, preprocessing the database into a data structure has two advantages in this setting: Besides the speedup in queries, which is the usual motivation for data structures, it allows us to avoid the privacy cost of asking

many queries. To deal with the high dimensionality of string data, we use insights from string algorithms, along with a filtering of rare patterns at various steps throughout the construction algorithm. We combine this with a novel differentially private algorithm for counting on trees. The result is a differentially private algorithm which produces a trie of $O(n\ell^2)$ nodes, where each node has a noisy count. For ϵ -differential privacy, we show that the maximum error of these noisy counts is $O(\ell)$, up to polylogarithmic factors in the problem parameters, for both DOCUMENT COUNT and SUBSTRING COUNT. Strings which are not present in the trie have a count which lies below the error bound, with high probability in the coin-flips of the algorithm. Surprisingly, this almost matches the discussed lower bound for computing the SUBSTRING COUNT of *one* pattern, even though the data structure can be used to compute the count of *all possible* patterns. For (ϵ, δ) differential privacy, we show how to reduce the error for DOCUMENT COUNT by a factor $\sqrt{\ell}$.

As an immediate application of our work, we get improved results for problems which have been extensively studied under differential privacy from a practical perspective: namely, the problem of *frequent sequential pattern mining* [7, 29, 11, 27, 24], or the very similar problem of *q-gram extraction* [25, 10], where a *q-gram* is a pattern of a fixed length q . The goal in both of these problems is to analyze a collection of strings and report patterns that either occur in many strings in the collection (i.e., have a high document count) or appear often as a substring in the collection (i.e., have a high substring count). The private algorithms in these papers have been used for analysing transit data [11] and publishing genome data [24]. However, existing papers do not give any theoretical guarantees on the accuracy of their algorithms in terms of the problem parameters. As an application of our data structures for DOCUMENT COUNT or SUBSTRING COUNT, we can solve these problems by reporting all patterns with a noisy count above a given threshold. We show that the additive error achieved by our ϵ -differentially private algorithm is tight for this problem formulation, up to poly-logarithmic factors. Thus, the problem we study is a natural generalization of these problems. In the full version of this paper [5], we argue that our tight bounds yield a polynomial improvement of the error compared to prior approaches.

1.1 Setup and Results

In this work, our database is a collection $\mathcal{D}$ of documents (which we sometimes call strings) of length at most ℓ drawn from an alphabet Σ of size $|\Sigma|$. That is, we have $\mathcal{D} = S_1, \dots, S_n$, where $S_i \in \Sigma^{[1, \ell]}$ for all $i = 1, \dots, n$. Thus, the *data universe* is $X = \Sigma^{[1, \ell]}$ and any database is an element of X^n . We call two databases $\mathcal{D} \in X^n$ and $\mathcal{D}' \in X^n$ *neighboring* if there exists an $S'_i \in \Sigma^{[1, \ell]}$ such that $\mathcal{D}' = S_1, \dots, S_{i-1}, S'_i, S_{i+1}, \dots, S_n$ for some i (that is, document S_i has been replaced with document S'_i). We also denote the (symmetric) neighboring relation as $\mathcal{D} \sim \mathcal{D}'$.

DEFINITION 1 (DIFFERENTIAL PRIVACY). *An algorithm A with output range $\text{range}(A)$ is (ϵ, δ) -differentially private, if for all neighboring $\mathcal{D}$ and $\mathcal{D}'$ and any set $U \subseteq \text{range}(A)$, we have*

$$\Pr[A(\mathcal{D}) \in U] \leq e^\epsilon \Pr[A(\mathcal{D}') \in U] + \delta.$$

For $\delta = 0$, the property from Definition 1 is also called *ϵ -differential privacy* or *pure differential privacy*. For $\delta > 0$,

it is also called *approximate differential privacy*.

Let P and S be strings over an alphabet Σ . Given a database $\mathcal{D} = S_1, \dots, S_n$, we want to build a data structure to efficiently answer the following types of queries while preserving differential privacy:

DOCUMENT COUNT(P): Return the number $\text{count}_1(P, \mathcal{D})$ of documents in $\mathcal{D}$ that contain P at least once.

SUBSTRING COUNT(P): Return the total number $\text{count}(P, \mathcal{D})$ of occurrences of P in the documents in $\mathcal{D}$.

EXAMPLE 1. Consider the database $\mathcal{D} = \{aaaa, abe, absab, babe, bee, bees\}$ and the string $P = ab$. Then $\text{count}_1(P, \mathcal{D}) = 3$, $\text{count}(P, \mathcal{D}) = 4$.

To ensure privacy, we need to introduce an error. We say that a data structure for **DOCUMENT COUNT** has an *additive error* α , if for any query pattern $P \in \Sigma^{[1, \ell]}$, it produces a value $\text{count}_1^*(P, \mathcal{D})$ such that $|\text{count}_1(P, \mathcal{D}) - \text{count}_1^*(P, \mathcal{D})| \leq \alpha$. Similarly, a data structure for **SUBSTRING COUNT** has an *additive error* α , if for any query pattern $P \in \Sigma^{[1, \ell]}$, it produces a value $\text{count}^*(P, \mathcal{D})$ such that $|\text{count}(P, \mathcal{D}) - \text{count}^*(P, \mathcal{D})| \leq \alpha$.

Main results.

In this paper, we give new ϵ -differentially private and (ϵ, δ) -differentially private data structures for **DOCUMENT COUNT** and **SUBSTRING COUNT**. We obtain new solutions to frequent substring mining as a corollary. We stress that for all of our data structures, the *construction algorithm* satisfies differential privacy, so the resulting data structures can be queried ad libitum without further privacy loss. We first state our result for ϵ -differential privacy, which gives both a data structure for **DOCUMENT COUNT** and for **SUBSTRING COUNT** with an additive error at most $O(\ell \cdot \text{polylog}(n\ell + |\Sigma|))$ with high probability. The data structure output by both Theorems 1 and 2 is a trie with $O(n\ell^2)$ nodes and a noisy count for each node. Patterns which are not present in the trie have a count which lies below the error bound with high probability.

THEOREM 1. *There exists an ϵ -differentially private algorithm, which can process any database $\mathcal{D} = S_1, \dots, S_n$ of documents in $\Sigma^{[1, \ell]}$ and with high probability outputs a data structure for **SUBSTRING COUNT** (resp. **DOCUMENT COUNT**) with additive error $O(\epsilon^{-1} \ell \log \ell (\log^2(n\ell) + \log |\Sigma|))$. The data structure can be constructed in $O(n^2 \ell^4)$ time and space, stored in $O(n\ell^2)$ space, and answer queries in $O(|P|)$ time.*

For **DOCUMENT COUNT**, we can get an (ϵ, δ) -differentially private data structure whose error is better by roughly a factor $\sqrt{\ell}$.

THEOREM 2. *There exists an (ϵ, δ) -differentially private algorithm, which can process any database $\mathcal{D} = S_1, \dots, S_n$ of documents in $\Sigma^{[1, \ell]}$ and with high probability outputs a data structure for **DOCUMENT COUNT** with additive error $O(\epsilon^{-1} \sqrt{\ell} \log(1/\delta) \log \ell (\log(n\ell) + \sqrt{\log |\Sigma| \log \log \ell}))$. The data structure can be constructed in $O(n^2 \ell^4)$ time and space, stored in $O(n\ell^2)$ space, and answer queries in $O(|P|)$ time.*

We can use the data structure to output frequent substrings in a simple way: For a given threshold τ , we traverse the trie

and output every string corresponding to a node with a noisy count at least τ . Note that the error of the approximate count propagates to the threshold: Assuming the error of the approximate counts in the trie is at most α , we can say the following: For any string we output, its true count is at least $\tau - \alpha$, and we will find all strings with a count at least $\tau + \alpha$. However, patterns whose count lies within the interval $[\tau - \alpha, \tau + \alpha]$ may be output or not.

In the full version [5], we give a lower bound which states that the bound from Theorem 1 is tight up to $\text{polylog}(n\ell)$ terms. We also give an almost linear time algorithm for computing **DOCUMENT COUNT** and **SUBSTRING COUNT** for q -grams with (ϵ, δ) -differential privacy. The error guarantees of that algorithm match the bounds of our main theorems.

In the full version, we also develop a new technique to differentially privately compute count functions on a tree, which may be of independent interest. We motivate this using two examples. First, let T be a tree, where every leaf corresponds to an element in the universe. The count of every leaf is the number of times the element appears in a given database, and the count of a node is the sum of the counts of the leaves below. This model captures any hierarchical composition of data items (i.e., by zip code, area, state), and the problem of computing the node counts has been studied under differential privacy (e.g. [29, 20]). As noted in [20], it can be solved via a reduction to differentially private range counting over the leaf counts. For ϵ -differential privacy, the binary tree mechanism by Dwork et al. [15] gives an error of roughly $O(\log^2 u)$ for this problem, where u is the size of the universe. The range counting problem under (ϵ, δ) -differential privacy has also been extensively studied recently [12, 22, 1, 8, 9, 2]. As a second example, consider again a tree whose leaves are the items of the universe; however, we additionally have a *color* for every item. Now, the goal is to compute for every node the *number of distinct colors* of elements present in the leaves below the node. We call this problem the *colored tree counting problem*. This is similar to the *colored range counting problem*, which is well-studied in the non-private setting [23, 19, 18], and was also recently studied with differential privacy under the name of “counting distinct elements in a time window” [21].

We give an ϵ -differentially private algorithm that can solve the colored tree counting problem with error $O(\log^2 u \log h)$, where h is the height of the tree. In general, our algorithm can construct a differentially private data structure for any counting function that is (i) monotone, in the sense that any node’s count cannot be larger than the sum of the counts of its children, and (ii) has bounded L_1 -sensitivity on the leaves, i.e., the true counts of the leaves do not differ too much on neighboring databases (see Definition 2 for a formal definition of sensitivity). We show an improvement for (ϵ, δ) -differential privacy, if additionally the sensitivity of the count function for every node is small.

1.2 Technical Overview

Simple approach. Before we introduce our algorithm, we present a simple trie-based approach (a similar strategy was adopted in previous work, e.g. [7, 29, 10, 11, 25, 24]). For simplicity, we focus on **SUBSTRING COUNT**. A private trie for the database is constructed top-down. Starting at the root, we create a new child node for every letter in the alphabet, connected by an edge labeled with the corresponding letter. Then, for each leaf, we compute how many times the corre-

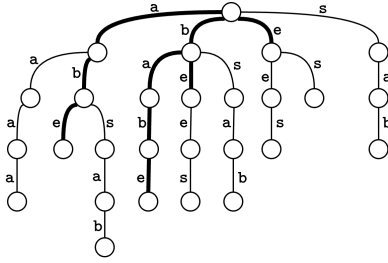


Figure 1: The trie of all suffixes of the strings in database $\mathcal{D}$ from Example 1. The paths corresponding to all suffixes of the string babe are in bold.

sponding letter occurs in $\mathcal{D}$ and add noise to this count. If the noisy count exceeds a certain threshold, we expand this node by adding a child for every letter in the alphabet. We continue expanding the trie from the newly added leaves, but in later rounds, when computing counts for a leaf, instead of considering a single letter, we count how many times the string obtained by concatenating the labels of the edges on the root-to-leaf path occurs as a substring in $\mathcal{D}$.

We repeat this process until no noisy count exceeds the threshold. The main advantage of this top-down approach is to avoid considering the entire universe by excluding patterns that are not frequent as soon as a prefix is not frequent. To make the algorithm ϵ -differentially private, the noise we add has to scale approximately with the L_1 -sensitivity of the true counts. The L_1 -sensitivity is here defined as how much the true counts of the nodes in the trie can differ in total if we replace a document in the database. Assume $\mathcal{D}' = \mathcal{D} \setminus \{S\} \cup \{S'\}$. Then, the count of a node is different on $\mathcal{D}$ and $\mathcal{D}'$ if and only if the string represented by the node is a substring of S or S' (or both). However, S and S' can each have up to $\Omega(\ell^2)$ different substrings. Thus, this approach yields an error of $\Omega(\ell^2)$.

Idea for improvement. An important insight is that while there are $\Omega(\ell^2)$ nodes in the trie that can be influenced by S and S' , these nodes can lie on at most 2ℓ different root-to-leaf paths, namely, the paths corresponding to the suffixes of S and S' . See Figure 1. Our idea is to leverage this property. For this, we make use of the *heavy path decomposition* [28] of a tree T , which partitions T into disjoint *heavy paths* and has the property that any root-to-leaf path in T crosses at most a logarithmic number (in the size of T) of such heavy paths. We will use these heavy paths to add noise in a way such that the error depends on the number of heavy paths containing nodes with a count that is different on $\mathcal{D}$ and $\mathcal{D}'$.

However, there are two main challenges. We cannot construct the heavy path decomposition during the top-down construction above, since this requires knowing the size of the subtrees before they have been explored, but at the same time, we need the noisy counts to decide which nodes to expand further. Alternatively, we could first construct the full trie of the universe (all possible substrings of length ℓ) and then compute the heavy path decomposition. But in this case, the size of the trie would be $|\Sigma|^\ell$, so the number of heavy paths on a root-to-leaf path would only be bounded by $\ell \log \Sigma$, and therefore, the ℓ root-to-leaf paths which contain nodes of different counts in $\mathcal{D}$ and $\mathcal{D}'$ can cross up to $\Omega(\ell^2 \log \Sigma)$ heavy paths.

Our algorithm. We now give an overview of our full algorithm. As a first step, we reduce the universe size from $|\Sigma|^\ell$ to $n^2 \ell^3$, by computing a set of *candidate frequent strings* in a differentially private way. Recall that we need substrings not occurring in our database $\mathcal{D}$ to appear in our candidate set with non-zero probability. We show how to efficiently obtain this without exhaustively considering all such substrings.

We iteratively construct a collection $\mathcal{P} = \mathcal{P}_{2^0}, \dots, \mathcal{P}_{2^{\lceil \log \ell \rceil}}$ of sets of strings, such that the strings in $\mathcal{P}_{2^k}$ have length 2^k , as follows. We first construct $\mathcal{P}_{2^0}$ by computing noisy counts for all letters in the alphabet, and keep only those with a noisy count above a certain threshold. To construct $\mathcal{P}_{2^k}$, we compute noisy counts of all strings obtained by concatenating two strings from $\mathcal{P}_{2^{k-1}}$, again keeping only those with a noisy count above the threshold. We show that the collection only contains substrings from $\mathcal{D}$ with high probability. Note that a document S can have at most ℓ substrings of any fixed length, and we consider a logarithmic number of lengths. Therefore, our noise needs to scale with $\ell \log \ell$. We then use the following observation: any string of a given length m which is not a power of two has an overlapping prefix and suffix whose length is a power of two. Our final candidate set consists of the collection $\mathcal{P}$ together with all strings covered by a prefix and suffix from $\mathcal{P}$.

Next, we build a trie from our candidate set and decompose it into heavy paths. For every heavy path, we compute (i) a differentially private noisy count of its root, and (ii) a differentially private estimate of the *prefix sums of the difference sequence of the counts* on the path going down. Specifically, for every node on the heavy path, we get an approximation of the *difference* between its count and the count of the root. We can compute (ii) with low noise using a generalized variant of the binary tree mechanism [15]. We show that since any root-to-leaf path can cross at most $\log(n^2 \ell^3) = O(\log(n\ell))$ heavy paths, and a document S can only influence counts on the at most ℓ paths corresponding to its suffixes, this can be done with an error which is $O(\ell \cdot \text{polylog}(n\ell))$. This strategy gives our main result of Theorem 1.

The improvement for q -grams stems from two observations: 1) since we only want to compute the counts for patterns of a fixed length, we do not need to compute the full set of candidate strings and can significantly simplify the algorithm, and 2) for (ϵ, δ) -differential privacy, we show that we can avoid computing noisy counts for patterns whose true count is 0, without violating privacy.

1.3 Roadmap

In Section 2, we collect some background on string data structures and differential privacy. In Section 3, we present the construction algorithm for our ϵ -differentially private data structure for SUBSTRING COUNT. In Section 4, we sketch a faster construction algorithm for SUBSTRING COUNT restricted to q -grams with (ϵ, δ) differential privacy.

2. PRELIMINARIES

In this work, we use $\log$ to denote the binary logarithm and $\ln$ to denote the natural logarithm. We use $[a, b]$ to denote the interval of integers $\{a, a+1, a+2, \dots, b-1, b\}$.

A string S of length $|S| = \ell$ is a sequence $S[0] \dots S[\ell-1]$ of ℓ characters drawn from an alphabet Σ of size $|\Sigma|$. The string $S[i] \dots S[j]$, denoted $S[i, j]$, is called a *substring* of

S ; $S[0, j-1]$ and $S[i, \ell-1]$ are a *prefix* and *suffix* of S , respectively. We say that a string P *occurs* in a string S iff there exists an i such that $S[i, i+|P|-1] = P$. We use $\Sigma^{[a,b]}$ to denote all strings S over Σ which satisfy $a \leq |S| \leq b$.

A *trie* for a collection of strings $\mathcal{C} = S_1, \dots, S_n$, denoted $T_{\mathcal{C}}$, is a rooted labeled tree, such that: (1) The label on each edge is a character of one or more S_i . (2) Each string in $\mathcal{C}$ is represented by a path in $T_{\mathcal{C}}$ going from the root down to some node (obtained by concatenating the labels on the edges of the path). (3) Each root-to-leaf path represents a string from $\mathcal{C}$. (4) Common prefixes of two strings share the same path maximally. For a node $v \in T_{\mathcal{C}}$, we let $\text{str}(v)$ denote the string obtained from concatenating the labels on the path's edges from the root to v .

A *compacted trie* is obtained from $T_{\mathcal{C}}$ by dissolving all nodes except the root, the branching nodes, and the leaves, and concatenating the labels on the edges incident to dissolved nodes to obtain *string* labels for the remaining edges. The compacted trie can be represented in $O(|\mathcal{C}|)$ space and computed in $O(|\mathcal{C}|)$ time [26].

DEFINITION 2 (L_p -SENSITIVITY). *Let f be a function $f : \chi^n \rightarrow \mathbb{R}^k$ for some universe χ . The L_p -sensitivity of f is defined as $\max_{\mathcal{D} \sim \mathcal{D}'} \|f(\mathcal{D}) - f(\mathcal{D}')\|_p$.*

The sensitivity of a function is a measure of how much its output can differ between neighboring databases. The *Laplace Mechanism* [14] computes the output of a function plus Laplace noise with a scale depending on ϵ and the L_1 -sensitivity of the function. Its properties are summarized in the following corollary.

COROLLARY 1. *Let f be a function $f : \chi^n \rightarrow \mathbb{R}^k$ for some universe χ with L_1 -sensitivity at most Δ_1 . Then there exists an ϵ -differentially private algorithm A which for any $\mathcal{D} \in \chi^n$ outputs $A(\mathcal{D})$ satisfying $\|A(\mathcal{D}) - f(\mathcal{D})\|_{\infty} \leq \epsilon^{-1} \Delta_1 \ln(k/\beta)$ with probability at least $1 - \beta$.*

Lemma 1 is due to Dwork and Lei [13] and shows how privacy degrades when combining multiple differentially private algorithms. For ϵ -differential privacy, this composition lemma is tight in the worst case (for (ϵ, δ) -differential privacy, tighter results are known, but would only save logarithmic factors in this work).

LEMMA 1. *Let A_1 be an (ϵ_1, δ_1) -differentially private algorithm $\chi^n \rightarrow \text{range}(A_1)$ and A_2 an (ϵ_2, δ_2) -differentially private algorithm $\chi^n \times \text{range}(A_1) \rightarrow \text{range}(A_2)$. Then $A_1 \circ A_2$ is $(\epsilon_1 + \epsilon_2, \delta_1 + \delta_2)$ -differentially private.*

3. CONSTRUCTION ALGORITHM

In this section, we show how to construct our data structure. For simplicity, we focus on SUBSTRING COUNT. The solution for DOCUMENT COUNT is similar. We make the following key observation.

OBSERVATION 1. *For any document $S \in \mathcal{D}$ and any length $m \leq \ell$, the cumulative count $\sum_{P \in \Sigma^m} \text{count}(P, S)$ of all length- m substrings of S is at most ℓ .*

This observation holds because, since the length of S is bounded by ℓ , S has at most $\ell - m + 1 \leq \ell$ fragments of length m (one for each possible starting position). Thus the

total number of occurrences of its substrings of length m is bounded by the same quantity. Observation 1 implies the following.

COROLLARY 2. *The L_1 -sensitivity of $(\text{count}(P, \mathcal{D}))_{P \in \Sigma^m}$ is bounded by 2ℓ , for any $m \leq \ell$.*

Our algorithm has 6 main steps.

Step 1. We run an ϵ -differentially private algorithm to compute a set $\mathcal{C}$ of *candidate frequent* strings such that the true count in $\mathcal{D}$ of any string not included in $\mathcal{C}$ is small, and the set $\mathcal{C}$ is not too large.

Step 2. We build the trie $T_{\mathcal{C}}$ of the candidate set $\mathcal{C}$, and construct a *heavy path decomposition* of $T_{\mathcal{C}}$.

Step 3. For every root r of a heavy path, we compute a differentially private estimate of $\text{count}(\text{str}(r), \mathcal{D})$.

Step 4. For every heavy path $p = v_0, v_1, \dots, v_{|p|-1}$ with root $r = v_0$, we consider the *difference sequence* of counts along the path: i.e., for a node v_i on the path and its parent v_{i-1} , the i th element in the difference sequence is given by $\text{count}(\text{str}(v_i), \mathcal{D}) - \text{count}(\text{str}(v_{i-1}), \mathcal{D})$, for $i = 1, \dots, |p| - 1$. We use the binary tree mechanism [15] to compute a differentially private estimate of all *prefix sums*, sums_p^* , of each of the difference sequences.

Step 5. We compute the noisy count of every node v in $T_{\mathcal{C}}$, thus of every substring in $\mathcal{C}$: For every node v_i on a heavy path $v_0, v_1, \dots, v_{|p|-1}$, we set $\text{count}^*(\text{str}(v_i), \mathcal{D}) = \text{count}^*(\text{str}(v_0), \mathcal{D}) + \text{sums}_p^*[i]$.

Step 6. We finally prune $T_{\mathcal{C}}$ removing the subtrees with sufficiently small noisy counts, and return the pruned trie as our ϵ -differentially private data structure.

In the rest of the section, we detail this approach.

3.1 Step 1: Computing a Candidate Set

As a first step, we show how to compute a candidate set $\mathcal{C}$ satisfying the following lemma.

LEMMA 2. *Let $\mathcal{D} = S_1, \dots, S_n$ be a database of documents. For any $\epsilon > 0$ and $0 < \beta < 1$ there exists an ϵ -differentially private algorithm, which computes a candidate set $\mathcal{C} \subseteq \Sigma^{[1, \ell]}$ enjoying the following two properties with probability at least $1 - \beta$:*

- $\text{count}(P, \mathcal{D}) = O(\epsilon^{-1} \ell \log \ell \log(\max\{\ell^2 n^2, |\Sigma|\}/\beta))$ for any $P \in \Sigma^{[1, \ell]}$ not in $\mathcal{C}$
- $|\mathcal{C}| \leq n^2 \ell^3$

We compute the candidate set as follows.

- 1.1 Inductively construct sets $\mathcal{P}_{2^0}, \dots, \mathcal{P}_{2^j}$, with $j = \lfloor \log \ell \rfloor$, where $\mathcal{P}_{2^k}$ contains only strings of length 2^k which appear sufficiently often as substrings in $\mathcal{D}$ according to noisy counts;
- 1.2 Compute, for each $m \leq \ell$ which is not a power of 2, the set $\mathcal{C}_m$ of all strings P of length m whose length- 2^k prefix and suffix are in $\mathcal{P}_{2^k}$, where $k = \lfloor \log m \rfloor$;
- 1.3 The set $\mathcal{C}$ is finally obtained as the union of all the sets computed in Steps 1.1 and 1.2.

More precisely, in Step 1.1 the algorithm first computes a noisy count of all letters via the Laplace mechanism (Corollary 1). From those, it computes the set $\mathcal{P}_{2^0}$ of letters whose noisy counts are above a threshold τ which is strictly larger

than the error bound of the Laplace mechanism. Thus, with high probability, we only keep letters appearing in $\mathcal{D}$, i.e., at most $n\ell$. It then constructs each of the sets $\mathcal{P}_{2^k}$ from $\mathcal{P}_{2^{k-1}}$ by computing noisy counts for all strings obtained by concatenating two strings from $\mathcal{P}_{2^{k-1}}$ and pruning those with a noisy count lower than τ . We maintain $|\mathcal{P}_{2^k}| \leq n\ell$ for all k .

EXAMPLE 2. Consider the database $\mathcal{D} = \{aaaa, abe, absab, babe, bee, bees\}$ from Example 1. The exact versions, $\mathcal{P}_{2^0}^\times, \mathcal{P}_{2^1}^\times, \mathcal{P}_{2^2}^\times$, of the sets $\mathcal{P}_{2^0}, \mathcal{P}_{2^1}, \mathcal{P}_{2^2}$ (i.e. the sets computed according to exact rather than noisy counts) with a frequency threshold $\tau = 1$ are

$$\begin{aligned}\mathcal{P}_{2^0}^\times &= \{a, b, e, s\} \\ \mathcal{P}_{2^1}^\times &= \{aa, ab, ba, be, bs, ee, es, sa\} \\ \mathcal{P}_{2^2}^\times &= \{aaaa, absa, babe, bees, bsab\}\end{aligned}$$

The noisy, ϵ -differentially private sets $\mathcal{P}_{2^0}, \mathcal{P}_{2^1}, \mathcal{P}_{2^2}$ that are actually computed by the construction algorithm may contain (with sufficiently low probability) substrings that do not occur at all in $\mathcal{D}$; or conversely, some substrings occurring in $\mathcal{D}$ may be missing from the sets. A possible outcome of a noisy computation of these sets for $\mathcal{D}$ is

$$\begin{aligned}\mathcal{P}_{2^0} &= \{a, b, e, s\} \\ \mathcal{P}_{2^1} &= \{aa, ab, ba, be, bs, ee, sa\} \\ \mathcal{P}_{2^2} &= \{aaaa, absa, babe, bsab, aaab\}\end{aligned}$$

Note that the string es is missing from $\mathcal{P}_{2^1}$ although it occurs once in $\mathcal{D}$, and consequently, $bees$ is missing from $\mathcal{P}_{2^2}$; conversely, $aaab$ is in $\mathcal{P}_{2^2}$ although it does not occur in $\mathcal{D}$.

The noisy counts for the strings in $\mathcal{P}_{2^0}, \dots, \mathcal{P}_{2^j}$ can be obtained via the Laplace mechanism from their exact counts, which in turn, can be computed efficiently by building a concatenation data structure [6] over the suffix tree of the strings in $\mathcal{D}$ [17]. Details are in [5].

Step 1.2, that is, computing candidate sets for strings whose lengths are not powers of 2, can be done as a post-processing step using sets $\mathcal{P}_{2^0}, \dots, \mathcal{P}_{2^j}$, thus it does not entail any further privacy loss. For any $2^k < m < 2^{k+1}$, the set $\mathcal{C}_m$ contains all strings of length m having their length- 2^k prefix and suffix in $\mathcal{P}_{2^k}$, regardless of whether they occur in $\mathcal{D}$. Note that $|\mathcal{C}_m| \leq n^2 \ell^2$ for each m .

EXAMPLE 3. Consider the same database $\mathcal{D}$ and sets $\mathcal{P}_{2^0}, \mathcal{P}_{2^1}, \mathcal{P}_{2^2}$ as in Example 2. The set $\mathcal{C}_3$ is

$$\mathcal{C}_3 = \{aaa, aab, aba, abe, abs, baa, bab, bee, bsa, eee, saa, sab\}$$

$\mathcal{C}_3$ contains all possible strings of length 3 whose length-2 prefix and suffix are in $\mathcal{P}_{2^1}$; these strings can be obtained by taking all ordered pairs of strings $Q_1, Q_2 \in \mathcal{P}_{2^1}$ such that the length-1 suffix of s_1 is equal to the length-1 prefix of s_2 : in this case, we say that Q_1 and Q_2 have a suffix/prefix overlap of length 1. Note that many strings of $\mathcal{C}_3$ (e.g. aba) do not occur at all in $\mathcal{D}$ although, in this example, all strings of $\mathcal{P}_{2^1}$ do. This is a normal behavior of our construction algorithm, and it is crucial to guarantee ϵ -differential privacy.

The set $\mathcal{C}_5$ is obtained from $\mathcal{P}_{2^2}$ by taking all ordered pairs of strings with a suffix/prefix overlap of length 3: we have

$$\mathcal{C}_5 = \{aaaaa, aaaab, absab\}$$

In general, the strings of a set $\mathcal{C}_m$ with $2^k < m < 2^{k+1}$ can be obtained by computing all pairs $Q_1, Q_2 \in \mathcal{P}_{2^k}$ such that the

length- $(2^{k+1} - m)$ suffix of Q_1 is equal to the length- $(2^{k+1} - m)$ prefix of Q_2 . These pairs can be computed efficiently by constructing an LCE data structure [4]: details are in [5].

The complete candidate set $\mathcal{C}$ is finally obtained as the union of the sets $\mathcal{P}_{2^j}$ and $\mathcal{C}_m$ constructed in Steps 1.1 and 1.2. The above algorithm implies Lemma 3.

LEMMA 3. Given $\mathcal{D} = S_1, \dots, S_n$ a database over $\Sigma^{[1, \ell]}$, a set $\mathcal{C}$ satisfying the properties of Lemma 2 can be computed in time $O(n^2 \ell^3 \log \log(n\ell) + n^2 \ell^4)$ and space $O(n^2 \ell^4)$.

3.2 Steps 2 to 4: Heavy Paths and Difference Sequences

The next steps are to arrange $\mathcal{C}$ in a trie $T_{\mathcal{C}}$ and compute its heavy path decomposition. Note that the number of nodes $|T_{\mathcal{C}}|$ is bounded by the total length of strings in $\mathcal{C}$, which is in turn bounded by $n^2 \ell^4$. This step of the algorithm exploits the way the counts associated with nodes of $T_{\mathcal{C}}$ can vary in neighboring databases. A first key observation is as follows: Consider the counts of the string $\text{str}(v)$ associated with any node v of $T_{\mathcal{C}}$ in two neighboring databases $\mathcal{D}$ and $\mathcal{D}' = \mathcal{D} \setminus \{S\} \cup \{S'\}$. The difference of the counts of $\text{str}(v)$ in $\mathcal{D}$ and $\mathcal{D}'$ depends only on how many times $\text{str}(v)$ occurs in the strings S and S' by which $\mathcal{D}$ and $\mathcal{D}'$ differ:

OBSERVATION 2. For any node v in $T_{\mathcal{C}}$ and any two neighboring databases $\mathcal{D}$ and $\mathcal{D}' = \mathcal{D} \setminus \{S\} \cup \{S'\}$, $|\text{count}(\text{str}(v), \mathcal{D}) - \text{count}(\text{str}(v), \mathcal{D}')| = |\text{count}(\text{str}(v), S) - \text{count}(\text{str}(v), S')|$.

Let us now focus on how the counts computed in neighboring databases can vary along paths of $T_{\mathcal{C}}$. For any path $p = v_0, v_1, \dots, v_{|p|-1}$ in the trie $T_{\mathcal{C}}$, we define the *difference sequence* of count on p as the $(|p| - 1)$ -dimensional vector $\text{diff}_p(\mathcal{D})[i] = \text{count}(\text{str}(v_i), \mathcal{D}) - \text{count}(\text{str}(v_{i-1}), \mathcal{D})$ for $i = 1 \dots |p| - 1$: see Figure 3(a) for an example. Note that the counts associated to the nodes of a path are non-increasing when descending the path: this is because, if v_j is a descendant of v_i , then $\text{str}(v_i)$ is a prefix of $\text{str}(v_j)$, thus $\text{count}(\text{str}(v_i), S) \geq \text{count}(\text{str}(v_j), S)$. This allows us to prove Lemma 4 (the proof is in [5]).

LEMMA 4. Let $\mathcal{D}$ and $\mathcal{D}' = \mathcal{D} \setminus \{S\} \cup \{S'\}$ be two neighboring databases. Then, for any path p with root r in $T_{\mathcal{C}}$, $\|\text{diff}_p(\mathcal{D}) - \text{diff}_p(\mathcal{D}')\|_1 \leq \text{count}(\text{str}(r), S) + \text{count}(\text{str}(r), S')$.

We now pair these properties with the well-known *heavy path decomposition* of a tree T , defined as follows. Every edge is either *light* or *heavy*. There is exactly one heavy edge outgoing from every node except the leaves, defined as the edge to the child whose subtree contains the most nodes; ties are broken arbitrarily. The longest *heavy path* is obtained by following the heavy edges from the root of T to a leaf: note that all edges hanging off this path are light. The other heavy paths are obtained by recursively decomposing all subtrees hanging off a heavy path. We call the topmost node of a heavy path (i.e., the only node of the path which is not reached by a heavy edge) its *root*. The heavy path decomposition of a tree with N nodes can be constructed in $O(N)$ time and has the following important property [28]:

LEMMA 5. Given a tree T with N nodes and a heavy path decomposition of T , any root-to-leaf path in T contains at most $\lceil \log N \rceil$ light edges.

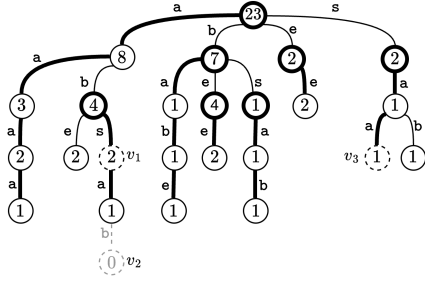


Figure 4: A possible output of the construction algorithm for the database of Examples 2 and 3, obtained by pruning the trie in Figure 2 according to noisy counts. Each node contains its noisy count. Dashed nodes contain erroneous counts: the exact count of v_1 is 1; v_2 has been pruned because of the noise, but its exact count is 1; the exact count of v_3 is 0, but it has not been pruned due to the added noise.

- Every string $P \in \Sigma^{[1, \ell]}$ which is not present in T^* was either: (i) not present in $\mathcal{C}$, in which case $\text{count}(P, \mathcal{D}) = O(\epsilon^{-1} \ell \log \ell (\log(n\ell/\beta) + \log|\Sigma|))$ by Lemma 2; or (ii) deleted in the pruning process of $T_{\mathcal{C}}$, in which case $\text{count}(P, \mathcal{D}) < 3\alpha = O(\epsilon^{-1} \ell \log \ell \log^2(n\ell/\beta))$.
- Any string $P \in \Sigma^{[1, \ell]}$ in T^* has a count at least $2\alpha - \alpha > 1$. Therefore, T^* has at most $O(n\ell^2)$ nodes.

To query the resulting data structure for a pattern P , we match P in the trie, and if there is a node v with $\text{str}(v) = P$, we return the approximate count $\text{count}^*(\text{str}(v), \mathcal{D})$ saved at v . If P is not present, we return 0. This requires $O(|P|)$ time. In summary, we have obtained Theorem 1.

4. EFFICIENTLY COUNTING Q-GRAMS

In this section, we give an intuition of why an (ϵ, δ) -differentially private data structure that answers q -gram counting queries in optimal time matching the error bounds of Theorem 2 (for DOCUMENT COUNT) and those of Theorem 1 (for SUBSTRING COUNT) can be constructed more efficiently than the general ϵ -differentially private data structure of Section 3. Specifically, we show how such a data structure can be computed in $O(n\ell(\log q + \log|\Sigma|))$ time and $O(n\ell)$ space:

Similar to the algorithm of Section 3, the construction algorithm restricted to q -grams begins by computing sets $\mathcal{P}_{2^0}, \mathcal{P}_{2^1}, \dots, \mathcal{P}_{2^j}$, with $j = \lfloor \log q \rfloor$, containing substrings of geometrically increasing length with sufficiently high noisy counts in $\mathcal{D}$. However, for the less stringent model of approximate differential privacy, we crucially observe that we can avoid considering substrings that do not appear in $\mathcal{D}$.

As a consequence, sets $\mathcal{P}_{2^0}, \mathcal{P}_{2^1}, \dots, \mathcal{P}_{2^j}$ can be computed by filtering the substrings occurring in $\mathcal{D}$ according to noisy counts, rather than generating a quadratic number of candidate strings at every step. This removes the computational bottleneck of the algorithm of Section 3.

The algorithm proceeds inductively starting from $\mathcal{P}_{2^0}$: It computes noisy counts for all letters of Σ occurring in $\mathcal{D}$ at least once, and filters out those whose noisy count is below an appropriately chosen threshold τ . For all subsequent sets $\mathcal{P}_{2^k}$, it computes noisy counts for all length- 2^k strings that

occur in $\mathcal{D}$ and whose length- 2^{k-1} prefix and suffix are in $\mathcal{P}_{2^{k-1}}$. Finally, it computes the set $\mathcal{P}_q$ of strings occurring in $\mathcal{D}$ whose length- 2^j prefix and suffix are in $\mathcal{P}_{2^j}$ and whose noisy counts exceed τ , and returns a compacted trie of $\mathcal{P}_q$ with a noisy counter for each node.

To perform these steps efficiently, we rely on the suffix tree of $\mathcal{D}$ [16] preprocessed to answer weighted ancestor queries in $O(1)$ time [3], and we represent sets $\mathcal{P}_{2^0}, \dots, \mathcal{P}_{2^j}$ implicitly in the suffix tree. Details are in [5].

5. CONCLUSION AND DISCUSSION

In this paper, we present the first differentially private data structures for two fundamental pattern matching problems, DOCUMENT COUNT and SUBSTRING COUNT. We show upper bounds on the error for these problems and complement them with lower bounds, showing that the accuracy of our data structures is tight up to $\text{polylog}(n\ell)$ terms. We extend this to get improved differentially private algorithms for frequent pattern mining and frequent q -gram mining. Finally, we present new data structures for counting on trees.

To achieve our results, we have developed several new techniques and combinations thereof for private data structures on strings and trees, including a new way to compute counting functions on tree structures using tree heavy path decomposition combined with the binary tree mechanism. Since such counting problems are basic primitives in computer science, our techniques are likely applicable in a wide range of areas. For example, possible applications could include string indexing for complex patterns (e.g., multiple patterns, regular expressions, and patterns with wildcards) and geometric data structure problems such as higher-dimensional range searching.

A key feature of our result is that we construct a differentially private data structure that we can ask many queries on without incurring further loss of privacy. While some differentially private data structures are known, we believe there is significant unexplored potential in designing differentially private solutions for classic data structure problems.

Our error bounds for pattern matching in the ϵ -differential privacy setting depend linearly on the string lengths and logarithmically on the number of strings. Thus, we expect our approach to perform well on databases containing many short or semilong strings, e.g., next-word suggestions or transit data. On the other hand, in scenarios with very long strings, e.g., genomic sequences, the error may be too large for practical applications. However, by our lower bound, this is unavoidable for ϵ -differential privacy if we want to protect entire strings. It is an interesting open question whether our error bounds for approximate differential privacy can be improved or shown to be tight.

For large databases, our construction time and space for general counting on strings may be infeasible. For q -grams, however, the construction time is almost linear. It is a relevant open problem whether the construction time and space can be improved in the general case.

In our presentation, we have focused on simplicity, and we have not attempted to optimize constants in our approach. However, when implementing our algorithms, one can likely obtain better error guarantees, e.g., by allocating the privacy budget differently across the algorithm's components.

6. REFERENCES

- [1] N. Alon, R. Livni, M. Malliaris, and S. Moran. Private PAC learning implies finite littlestone dimension. In *Proc. 51st STOC*, pages 852–860, 2019.
- [2] A. Beimel, K. Nissim, and U. Stemmer. Private learning and sanitization: Pure vs. approximate differential privacy. *Theory Comput.*, 12(1):1–61, 2016.
- [3] D. Belazzougui, D. Kosolobov, S. J. Puglisi, and R. Raman. Weighted ancestors in suffix trees revisited. In *Proc. 32nd CPM*, volume 191 of *LIPICs*, pages 8:1–8:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.
- [4] M. A. Bender and M. Farach-Colton. The LCA problem revisited. In *Proc. 4th LATIN*, volume 1776 of *Lecture Notes in Computer Science*, pages 88–94. Springer, 2000.
- [5] G. Bernardini, P. Bille, I. L. Gørtz, and T. A. Steiner. Differentially private substring and document counting. *Proc. ACM Manag. Data*, 3(2):95:1–95:27, 2025.
- [6] P. Bille, A. R. Christiansen, P. H. Cording, I. L. Gørtz, F. R. Skjoldjensen, H. W. Vildhøj, and S. Vind. Dynamic relative compression, dynamic partial sums, and substring concatenation. *Algorithmica*, 80(11):3207–3224, 2018.
- [7] L. Bonomi and L. Xiong. A two-phase algorithm for mining sequential patterns with differential privacy. In *Proc. 22nd CIKM*, pages 269–278, 2013.
- [8] M. Bun, C. Dwork, G. N. Rothblum, and T. Steinke. Composable and versatile privacy via truncated CDP. In *Proc. 50th STOC*, pages 74–86, 2018.
- [9] M. Bun, K. Nissim, U. Stemmer, and S. P. Vadhan. Differentially private release and learning of threshold functions. In *Proc. 56th FOCS*, pages 634–649, 2015.
- [10] R. Chen, G. Ács, and C. Castelluccia. Differentially private sequential data publication via variable-length n-grams. In *Proc. 19th CCS*, pages 638–649, 2012.
- [11] R. Chen, B. C. M. Fung, B. C. Desai, and N. M. Sossou. Differentially private transit data publication: a case study on the montreal transportation system. In *Proc. 18th KDD*, pages 213–221, 2012.
- [12] E. Cohen, X. Lyu, J. Nelson, T. Sarlós, and U. Stemmer. Optimal differentially private learning of thresholds and quasi-concave optimization. In *Proc. 55th STOC*, pages 472–482, 2023.
- [13] C. Dwork and J. Lei. Differential privacy and robust statistics. In *Proc. 41st STOC*, pages 371–380, 2009.
- [14] C. Dwork, F. McSherry, K. Nissim, and A. D. Smith. Calibrating noise to sensitivity in private data analysis. In *Proc. 3rd TCC*, volume 3876, pages 265–284. Springer, 2006.
- [15] C. Dwork, M. Naor, T. Pitassi, and G. N. Rothblum. Differential privacy under continual observation. In *Proc. 42nd STOC*, pages 715–724, 2010.
- [16] M. Farach. Optimal suffix tree construction with large alphabets. In *Proc. 38th FOCS*, pages 137–143. IEEE Computer Society, 1997.
- [17] M. Farach-Colton, P. Ferragina, and S. Muthukrishnan. On the sorting-complexity of suffix tree construction. *J. ACM*, 47(6):987–1011, 2000.
- [18] Y. Gao. Adaptive data structures for 2d dominance colored range counting. In *Proc. 18th WADS*, pages 460–473, 2023.
- [19] Y. Gao and M. He. Space efficient two-dimensional orthogonal colored range counting. In *Proc. 29th ESA*, pages 46:1–46:17, 2021.
- [20] B. Ghazi, P. Kamath, R. Kumar, P. Manurangsi, and K. Wu. On differentially private counting on trees. In *Proc. 50th ICALP*, volume 261, pages 66:1–66:18, 2023.
- [21] B. Ghazi, R. Kumar, J. Nelson, and P. Manurangsi. Private counting of distinct and k-occurring items in time windows. In *Proc. 14th ITCS*, pages 55:1–55:24, 2023.
- [22] H. Kaplan, K. Ligett, Y. Mansour, M. Naor, and U. Stemmer. Privately learning thresholds: Closing the exponential gap. In *Proc. 33rd COLT*, pages 2263–2285, 2020.
- [23] H. Kaplan, N. Rubin, M. Sharir, and E. Verbin. Efficient colored orthogonal range counting. *SIAM J. Comput.*, 38(3):982–1011, 2008.
- [24] T. Khatri, G. G. Dagher, and Y. Hou. Privacy-preserving genomic data publishing via differentially-private suffix tree. In *Proc. 15th EAI*, pages 569–584, 2019.
- [25] K. Kim, S. Gopi, J. Kulkarni, and S. Yekhanin. Differentially private n-gram extraction. In *Proc. 34th NeurIPS*, pages 5102–5111, 2021.
- [26] G. M. Landau, T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Proc. 12th CPM*, pages 181–192, 2001.
- [27] M. Nunez-del Prado, Y. Maehara-Aliaga, J. Salas, H. Alatrística-Salas, and D. Megías. A graph-based differentially private algorithm for mining frequent sequential patterns. *Applied Sciences*, 12(4), 2022.
- [28] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *J. Comput. Syst. Sci.*, 26(3):362–391, 1983.
- [29] J. Zhang, X. Xiao, and X. Xie. Privtree: A differentially private algorithm for hierarchical decompositions. In *Proc. SIGMOD Conference 2016*, pages 155–170, 2016.

Technical Perspective: Diva: Dynamic Range Filter for Var-Length Keys and Queries

Viktor Leis
Technical University of Munich
leis@in.tum.de

To accelerate data access, database systems have long relied on probabilistic filters, most notably Bloom filters, to avoid unnecessary I/O. These compact structures are highly effective for point queries: they answer “is key x present?” with one-sided error, i.e., with false positives but not false negatives.

This success does not carry over naturally to range predicates. Hashing destroys order, and once the workload shifts from membership to range emptiness – “is there any key in $[l, r]$?” – traditional hash-based filters cease to be a satisfactory solution. Range filters fill this gap by allowing systems to rule out empty intervals and skip disk pages, SSTables, or cloud objects. This is particularly valuable in LSM-based engines, where pruning across many runs or levels is essential to performance.

What makes this problem especially difficult is that the properties required in practice pull in different directions. A useful range filter should operate under a tight DRAM budget, support arbitrary query lengths, handle variable-length keys, admit updates and growth, and still remain fast both to construct and to query. The paper’s first contribution is a particularly clear articulation of this design space. It explains why all desirable guarantees cannot simultaneously hold in their strongest form for worst-case datasets, and uses this impossibility result not as a limitation but as guidance. Diva is explicitly designed for the common case that matters in real systems: realistic key distributions, variable-length keys and queries, and dynamic workloads. Thus, the work aims for completeness, but it does so with a careful understanding of what theory permits and what systems need.

The central idea behind Diva is elegant. It picks roughly one in a thousand keys from the sorted set and builds a cache-efficient trie from them that captures the coarse structure of the key space. Diva then compresses the keys between adjacent samples in a way that preserves order information without resorting to hashing. For each such group, Diva removes the longest common prefix that the trie already explains, and then truncates suffix bits while keeping a fixed-size middle fragment, or infix, that is still sufficient to distinguish keys in sorted order with low false-positive probability. This turns the sampled trie into a lightweight distribution model and the infixes into a compact, order-aware

representation of local detail. The effect is exactly what one would want: dense regions are represented at finer effective granularity, while sparse regions are represented more coarsely, allowing the filter to adapt naturally to both short and long range queries.

The engineering of the Infix Stores, which make up most of the overall space, is impressive. Diva stores these infixes using quotienting, compact bitmaps, and rank/select-based navigation, so that queries examine only a small number of slots. Diva supports insertions, deletions, and growth by overprovisioning Infix Stores, spacing runs evenly, shifting locally when needed, and splitting or merging stores as new keys are added to or removed from the trie. This combination of conceptual simplicity and systems-minded detail is rare.

The empirical results reinforce the paper’s claims. Across the evaluated datasets and workloads, Diva delivers a strong balance between false-positive rate, latency, construction cost, and update overhead. In end-to-end experiments inside WiredTiger, Diva speeds up empty range-query processing by as much as three orders of magnitude.

Overall, Diva is an excellent example of database systems research at its best. It starts from an important practical problem, identifies precisely why existing approaches fall short, and then introduces a design whose power comes not from unnecessary complexity, but from the right idea. The paper combines theory, algorithmic insight, and careful implementation to produce a range filter with strong real-world tradeoffs.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 2024 ACM 0001-0782/24/0X00 ...\$5.00.

Dynamic Range Filtering Beyond Worst-Case Bounds

Navid Eslami
University of Toronto

Ioana O. Bercea
KTH Royal Institute of Technology

Niv Dayan
University of Toronto

ABSTRACT

Range filters are compact data structures that answer approximate range emptiness queries. They are used in many domains, e.g., in key-value stores, to quickly rule out the existence of keys in a given query range and avoid having to search for them in storage. However, all existing range filters exhibit at least one of the following shortcomings: (1) they do not provide robust false positive rate and performance guarantees, (2) they do not support variable-length keys and query ranges, and (3) they do not allow dynamic operations such as insertions, deletions, or expansions.

We introduce Diva, the first range filter to address all the above challenges simultaneously. Diva learns the dataset's distribution by sampling keys and storing them in a cache-efficient trie. It compresses the keys in-between samples by removing their longest common prefix and truncating their suffixes while leaving enough bits in the middle (i.e., an infix) to allow differentiating between the keys in the sorted order. It stores infixes in constant-time dynamic data blocks, which it stretches and eventually splits to handle insertions and expansions. It processes a range query by traversing the trie and checking for the inclusion of at least one infix in the target query range.

Diva is the culmination of several years of research on filters from Orca Lab at the University of Toronto, in collaboration with KTH and Copenhagen University. This paper describes how Diva builds on this body of work, and how it addresses the limitations of prior art.

1. INTRODUCTION

What is a Filter? A filter is a memory-efficient probabilistic data structure that answers whether a query key exists in a given set. Its compactness allows it to fit in a higher level of the memory hierarchy than the set it represents, making it fast to query. A filter never returns a false negative, but may return a false positive with a probability called the *False Positive Rate (FPR)* that depends on its memory

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

©Copyright held by the owner/author(s). Publication rights licensed to the VLDB Endowment. This is a minor revision of the paper "Diva: Dynamic Range Filter for Var-Length Keys and Queries" published in PVLDB, Vol. 18, No. 11, ISSN 2150-8097. DOI: <https://doi.org/10.14778/3749646.3749664>.

footprint. Due to these qualities, filters are used in many applications to avoid redundant disk reads [22] and network hops [9] when a query happens to target a nonexistent key.

Range Filters and Applications. In contrast, a range filter accepts two keys as the end-points of a range and determines if at least one key in the set is in-between them. As with a traditional filter, a range filter does not return false negatives but may return false positives. Range filters were originally developed to support write-optimized database indexes such as LSM-trees [42], where queries must examine multiple candidate files, many of which may not contain any keys within a query's target range [59, 37, 41, 34, 52, 11, 25, 12]. Their applicability, however, is much broader: range filters also help prevent redundant I/Os to SQL tables [1] and B-tree indexes [25, 12], and they have applications in social web analytics [17] and distributed key-value stores [50].

Goals. The ideal general-purpose range filter must simultaneously support (G1) the lowest possible FPR under a stringent memory budget, (G2) range queries of any length, (G3) variable-length keys, (G4) dynamic modification operations such as insertions, deletions, and expansions, and the best possible (G5) query and (G6) construction performance. In addition, it should support point queries at least as well as traditional point filters (e.g., Bloom [8, 48], Cuckoo [43, 26], or Quotient filters [6, 45, 46]).

Design Contentions. Every existing range filter only fulfills a subset of these goals [1, 59, 37, 54, 27, 41, 34, 52, 11, 18, 25, 12]. In fact, almost none of them attain (G3) or (G4) [37, 54, 27, 41, 34, 52, 11, 18]. Moreover, Goswami et. al. have proven an information-theoretic lower bound on the memory footprint of range filters [30], stating that it is impossible to achieve (G1) and (G2) at the same time. However, this lower bound only holds for worst-case datasets, meaning that it may still be possible to achieve (G1) and (G2) for "common" datasets. As such, we pose the following research question: *Is it possible to design a range filter that simultaneously fulfills all six goals for common datasets?* This paper presents an affirmative answer.

Diva. We introduce Diva, the first range filter to support dynamic operations, variable-length queries and keys, and high performance, all at the same time. Diva learns the dataset's distribution by sampling keys and storing them in a cache-efficient trie. For all keys in-between two samples of the trie, Diva removes the longest common prefix. It also truncates their suffixes while keeping enough bits in the middle of each key (i.e., an infix) to differentiate them in most cases, thus achieving (G1). At the same time, the trie separates dense and sparse regions of the

Table 1: Definitions of terms and symbols.

Symbol	Definition
$q = [q_l, q_r]$	An inclusive query range.
ϵ	Target FPR, i.e., probability of a false positive.
L	Average key length.
$(a)_2$	Binary value represented by a .
T	Number of keys between two samples.
m_{shared}^i	Length of the longest common prefix of the i -th and $(i + 1)$ -th samples, in bits.
m_{infix}	Length of the infixes.
$m_{\text{redundant}}^i$	Number of redundant bits corresponding to the i -th and $(i + 1)$ -th samples.
α	Load factor of the Infix Stores.
x_q	Quotient of the infix x .
x_r	Remainder of the infix x .

key space. This allows for handling short range queries over densely populated regions and long range queries over sparse regions, thus meeting (G2). By discretizing all keys into fixed-length infixes without the use of hashing, Diva achieves (G3). This discretization also allows for storing the infixes of adjacent groups of keys within a constant time data structure called an Infix Store, fulfilling (G5). As Diva derives infixes without hashing and stores them in the original sorted order of the keys, it can be constructed using one sequential pass, making it the fastest range filter to construct and enabling (G6). To support dynamicity (G4), Diva builds on a complementary line of work from our lab and collaborators on resizable filters [20, 19, 32, 25, 12].

2. PROBLEM ANALYSIS

This section shows that no current range filter satisfies all of the six goals outlined in Section 1. We leave in-depth summaries of these filters to [25, 47].

Range Filtering Definitions. A range filter represents a set S of keys from a universe of size u . Given a range query of the form $q = [q_l, q_r]$, the filter checks if the range is empty, i.e., whether $q \cap S = \emptyset$, with an FPR of at most ϵ , where $0 < \epsilon < 1$. The top three rows of Table 1 summarize terms describing the range filtering problem throughout the paper.

Memory Lower Bound. A range filter is *Robust* if it guarantees an FPR of at most ϵ for any dataset. It is known that any robust range filter supporting queries of length up to R must use at least $\log_2 \frac{R}{\epsilon} - O(1)$ Bits per Key (BPK) [30], meaning that answering longer range queries requires more memory. Intuitively, this is because a robust range filter supporting longer queries must carry more information about which areas of the key space are empty. A non-robust range filter often drops all information about the keys' lower-order bits. This can lead to a high FPR when range queries predicate over these lower-order bits.

Robustly Achieving (G1), (G2), and (G3) is Impossible. In most applications, filters are allotted a stringent memory budget of 8-16 BPK to fit in memory (with a higher budget, one may as well store the full keys). If we rearrange the above lower bound in terms of range query length R and plug in 16 BPK as the memory budget and $\epsilon = 0.01$ as the target FPR, we find that a robust range filter can only answer range queries of length at most 512. Such short query lengths limit the applicability of the filter.

Table 2: Diva is the first range filter to simultaneously support variable-length queries and keys, as well as dynamicity.

Filter	Robust FPR (G1)	Semi-Robust FPR (G1)	Var.-Len. Queries (G2)	Var.-Len. Keys (G3)	Dynamic (G4)
SuRF			✓	✓	
Rosetta	✓	✓			
REncoder					
bloomRF					
Proteus					
SNARF		✓	✓		
Oasis+		✓	✓		
Grafite	✓	✓			
Memento	✓	✓			✓
Aeris	✓	✓			✓
Divia		✓	✓	✓	✓

The lower bound also implies that a robust range filter cannot support variable-length keys. With variable-length keys, there can be infinitely many possible keys within a range, meaning $R = \infty$. Plugging $R = \infty$ into the lower bound, we find that infinite memory is needed to support variable-length keys. In sum, it is impossible to attain all of (G1), (G2), and (G3) with a robust range filter.

Indeed, none of the existing robust range filters [37, 18, 25, 12] support variable-length queries (G2) or keys (G3), as they assume range query lengths bounded by R . For example, Memento filter [25], our lab's previous range filter, assumes keys are fixed-length and splits each into a prefix and a $\lceil \log_2 R \rceil$ -bit suffix. It hashes a key's prefix to store its suffix within a compact, approximate hash table. Since a range query longer than R spans multiple prefixes, Memento filter must query the hash table for each constituent prefix to answer the query. This causes the FPR to compound beyond ϵ and eventually approach 1 as the number of checks increases, since each check can yield a false positive. All other robust range filters also do more checks for longer range queries, leading to a higher FPR [37, 18, 12].

Semi-Robust FPR Guarantee. We observe that despite the above impossibility result, one can achieve (G1), (G2), and (G3) by providing a semi-robust FPR guarantee: *Any query must be answered with an FPR of at most ϵ when the dataset comes from a "well-behaved distribution."* Intuitively, a well-behaved distribution is one for which the cumulative distribution function is smooth. This property is commonly satisfied in practice since input datasets typically follow well-known, smooth distributions (e.g., Uniform, Normal, Zipfian, Power Law, Poisson). We formally define well-behaved distributions and show that Diva provides the above guarantee in Section 4 of our original paper [24].

Interestingly, SNARF [52] and Oasis+ [11] are learning-augmented range filters [36, 7] that fulfill (G1) and (G2) with the same semi-robust FPR guarantee. They do so by fitting a linear spline model to the keys' CDF. Using this model, they map each key to a bit in a large bitmap, setting it to 1. These filters handle variable-length queries by leveraging the monotonicity of this mapping and searching the bits corresponding to the query for a 1. There is no formalism or proof of the above FPR guarantee in their respective publications, though Diva's proof in Section 4 of our original paper [24] applies to them as well.

Nonetheless, SNARF and Oasis+ assume fixed-length keys when learning the data distribution and therefore fail to sat-

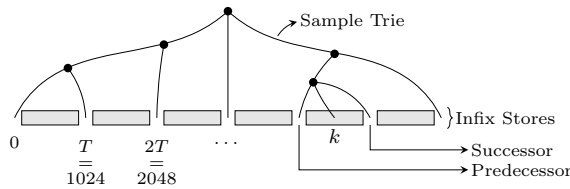


Figure 1: Diva learns the dataset’s distribution by storing every T -th key as a sample in a trie. It manages all keys between two adjacent samples in an Infix Store.

isfy (G3). Moreover, both incur high query latency due to their reliance on binary search, violating (G5). Although SNARF nominally supports insertions and deletions, these operations are prohibitively slow because its bitmap is encoded using the compact but static Elias-Fano scheme [23, 28]. Oasis+ does not support dynamic updates, as it prunes empty regions of the key space to improve its false positive rate. Consequently, neither filter satisfies (G4).

Challenges of Attaining Dynamicity (G4). Many existing range filters operate as Bloom filters in their core, hashing keys into a bitmap and setting bits from 0s to 1s [37, 54, 27, 41, 34]. As with standard Bloom filters, such filters cannot support deletes (by resetting a bit back to 0) or expansions (by remapping the 1s to a larger bitmap) without introducing false negatives [25, 12].

Other range filters utilize compact encoding schemes (e.g., succinct tries [59] or Elias-Fano [52, 11, 18]). Such formats are difficult to update, as they tightly pack data to avoid storing pointers and offsets. This necessitates changing their entire representation to make room for insertions.

Our prior papers, Memento filter and Aeris filter [25, 12], are the first range filters to support fast insertions, deletions, and resizability. They achieve this by building on our line of work on resizable filters [20, 19]. However, since they double in size to expand, they waste over 50% of their capacity right after expansion. Diva also builds on our prior work on resizable filters and pushes the envelope by allowing expansions by factors smaller than 2, as discussed in Section 3.4.

Contention between Supporting Variable-Length Queries (G2) and Query Speed (G5). It is an open question whether achieving fast operations while supporting variable-length range queries is possible. Grafite [18] and Memento filter [25] are the only filters that provide constant-time queries. They do so by localizing partitions of size R (i.e., the maximum range query length) of the key space to the same memory region. It is unclear how to achieve a similar localization with variable-length range queries. As such, Grafite and Memento filter do not fulfill (G2). All other filters that attain (G1) and (G2), i.e., SNARF [52] and Oasis+ [11], use predecessor search to handle queries, which requires super-constant time [4, 5]. In practice, these filters use binary search to compute predecessors. Diva alleviates this contention by employing a y -Fast trie [56] and ensuring that it fits in CPU caches, leading to faster predecessor searches.

Construction Speed (G6). Existing range filters use hashing [59, 37, 54, 27, 41, 34, 18, 25, 12] or floating-point operations [52, 11] extensively during construction, incurring high CPU and cache miss costs. In contrast, Diva avoids these operations and performs a single sequential pass over the data, thus supporting the fastest construction speed (G6).

Summary. Table 2 summarizes the goals each range filter meets. As shown, no preexisting range filter achieves (G1), (G2), (G3), and (G4) simultaneously. Table 2 also raises

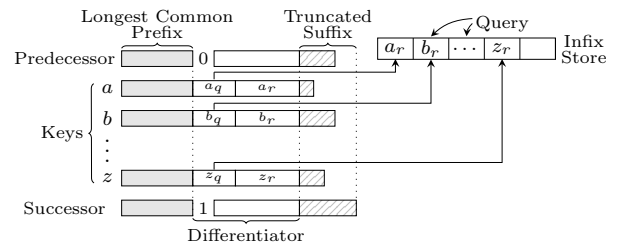


Figure 2: For all keys in-between a pair of adjacent samples in the trie, Diva removes the longest common prefix and truncates the longest possible suffix while still ensuring that there are enough bits left to distinguish between the keys.

the question of how performant a range filter can be to attain (G5) and (G6) while meeting (G1)-(G4).

3. DIVA

We present Diva: the first range filter to simultaneously support (G1) a low FPR and memory consumption, (G2) arbitrarily sized range queries, (G3) variable-length keys, (G4) dynamic updates, deletes, and resizability, and high (G5) query and (G6) construction performance. Diva picks every T -th key in the ordered key set as samples and stores them in a cache-efficient trie to approximate the key distribution. It inserts all keys between two samples into a compact data structure called an *Infix Store*, as shown in Figure 1. For all keys in each Infix Store, Diva removes the longest common prefix. It also homogenizes the keys’ lengths by truncating a suffix from each while leaving enough bits in the middle (i.e., an infix) to differentiate the keys and guarantee a given FPR. Figure 2 illustrates the truncation and mapping of keys into an Infix Store. To answer a range query, Diva checks for the absence of overlapping keys and infixes across its trie and the relevant Infix Stores, as discussed in Section 3.3. We describe a static version of Diva in Sections 3.1 to 3.3 and generalize it to dynamic sets in Section 3.4. The bottom part of Table 1 summarizes the terms used to describe Diva.

3.1 Sampling Keys and Deriving Infixes

The static variant of Diva requires the input to arrive in sorted order. This requirement is commonly satisfied in practice. For instance, many database tables and indices are structured as sorted column files [14, 2, 3, 29, 51], B-Trees [15, 49], or LSM-Trees [22, 16, 21]. If the input is unsorted, users must sort it before construction.

Diva samples every T -th key from the ordered key set and inserts it into its trie. The trie approximates the distribution of the keys. The reason is that adjacent samples lexicographically close to each other correspond to denser regions of the key space, while faraway samples correspond to sparser regions. Thus, the trie “learns” the data distribution. The smaller T is, the more accurate the approximate distribution becomes, yet the larger the trie’s memory footprint grows. We find that $T = 1024$ provides good all-round accuracy and performance with little memory overhead ($\approx 1\%$ of the filter’s memory). We structure the trie as a y -Fast trie [56] to achieve faster search times than traditional tries ($O(\log_2 L)$ time vs $O(L)$, where L is the search key’s length). We use Wormhole [57] as our y -Fast trie implementation.

Removing the Longest Common Prefix. Consider a pair of consecutive trie samples. We refer to them as the *Predecessor*

Predecessor and *Successor* of the keys between them. Figure 1 shows an example of predecessor and successor for a key k . For all keys in-between such a pair of samples, Diva removes their longest common prefix since it can be inferred from the trie and is thus redundant.

Keys in the denser regions of the key space have longer common prefixes. Therefore, removing the longest common prefixes saves more memory in these regions. As we will see shortly, Diva exploits this and encodes the keys in such regions at a higher resolution by also storing their lower-order bits, losing less information. This enables accurately answering fine-grained queries over dense regions while supporting coarse-grained queries over sparse regions.

Given the i -th and $(i + 1)$ -th samples as the predecessor and successor, we denote the length of their longest common prefix, in bits, as m_{shared}^i . Figure 3-A shows an example where the longest common prefix of the predecessor and successor is $m_{\text{shared}}^i = 4$ bits long. If the samples have different lengths, Diva treats the shorter one as having trailing zeros to match the length of the longer one.

Deriving Infixes. To further curb memory footprint, Diva also truncates suffixes for all keys between two samples. What remains of each key is called an *Infix* since it is a sequence of adjacent bits in the middle of the original key. All infixes in the filter have the same length, denoted by m_{infix} . Due to the removed common prefix, infixes represent less significant bits of the keys in dense regions of the key space and more significant bits in sparser ones. Truncating keys into fixed-length infixes discretizes the range between two samples while removing the variation in length of the keys. As we will see, Diva supports range queries over these infixes by correspondingly discretizing the query boundaries and checking for the inclusion of infixes between them. The fixed length of the infixes allows for quickly checking for inclusion using integer arithmetic.

The length of the infixes determines the filter’s FPR. In particular, Diva guarantees an FPR of ϵ by using infixes of length $m_{\text{infix}} = \lceil \log_2 \frac{2T}{\epsilon} \rceil$ bits. This guarantee holds for any dataset sampled from a “well-behaved” distribution. This property is suitable for a practical range filter since datasets in practice often follow distributions such as Normal or Zipfian. We prove this guarantee in Section 4 of our original paper [24]. As we show in Section 3.2, Diva succinctly encodes each infix using $3 + \lceil \log_2 \frac{1}{\epsilon} \rceil$ bits, yielding an FPR vs memory tradeoff on-par with state-of-the-art point filters.

Figure 3-A shows an example with keys of varying lengths and $m_{\text{infix}} = 7$. As $m_{\text{shared}}^i = 4$, Diva derives k ’s infix as the 7 bits following its first 4 bits, i.e., the string $(0101001)_2$.

Distinguishing Infixes. There is a common scenario in practice where infixes derived with the method described above have bits that do not contribute to filtering. Intuitively, if the considered predecessor and successor are still close to each other after removing their longest common prefix, the discretized key space between them becomes smaller than desired. As a result, infixes of keys in-between can become less distinguishable, reducing filtering accuracy. This phenomenon occurs when many consecutive bits after the first differentiating bit in the predecessor and successor are all 1s and 0s, respectively. In particular, in well-behaved distributions such as Normal and Zipfian, there is at least one such bit on average due to the randomness in the samples after their longest common prefix. If not removed, these bits can increase the FPR by $2\times$ compared to the case where in-

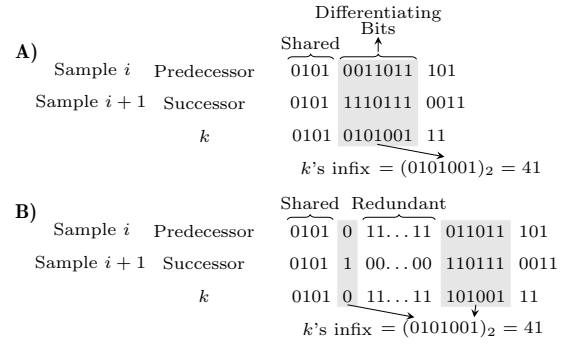


Figure 3: Diva derives key k ’s infix by removing the m_{shared}^i common prefix bits and the $m_{\text{redundant}}^i$ bits based on its predecessor and successor, and also truncating its suffix.

fixes are differentiated well. The predecessor and successor in Figure 3-B represent an example.

Diva prevents this by identifying the longest matching sequence of 1s from the predecessor and 0s from the successor after their first differentiating bit. We denote the length of this sequence for the i -th and $(i + 1)$ -th samples by $m_{\text{redundant}}^i$. Diva removes the bits corresponding to this sequence from each key between the samples. These bits are redundant, as they do not carry useful information for comparing the keys between the i -th and $(i + 1)$ -th samples beyond what the first differentiating bit provides. Diva derives the infix of each key as its first differentiating bit concatenated with its first $m_{\text{infix}} - 1$ bits after its redundant bits. Intuitively, each of these bits doubles the size of the discretized space between two samples, allowing to better differentiate the infixes in-between.

Diva is able to reconstruct the redundant bits removed from an infix. It does so by recomputing $m_{\text{redundant}}^i$ from its predecessor and successor in the trie. It adds $m_{\text{redundant}}^i$ bits between the infix’s first and second bits, each equal to the negation of its first bit. This results in the original sequence of bits in the key since the redundant bits of an infix always equal the negation of its first bit.

Figure 3-B shows an example derivation of a key k ’s infix with $m_{\text{infix}} = 7$. Here, the longest common prefix of the predecessor and successor is $m_{\text{shared}}^i = 4$ bits long. After removing the $m_{\text{redundant}}^i$ bits, Diva appends the next 6 bits in k (i.e., 101001) to the first differentiating bit (i.e., 0), yielding the infix $(0101001)_2 = 41$.

Infix Uniformity. The samples in the trie behave similarly to the boundaries of an equi-depth histogram. When the dataset’s distribution is well-behaved, the higher-order bits of the samples capture the overall shape of the distribution and the lower-order bits of the keys in-between behave like noise. Thus, the infixes of keys falling into the same bin are almost uniformly distributed. We formally prove this property in Section 4 of our original paper [24]. In the next section we leverage this property to design an efficient data structure for storing the infixes in-between a pair of samples.

3.2 Infix Stores

An Infix Store is a random-access array of slots that leverages the uniformity of the infixes in-between two samples to efficiently and succinctly store them. It utilizes techniques inspired by Cleary hash tables [13] and Rank-and-Select Quotient Filters [45].

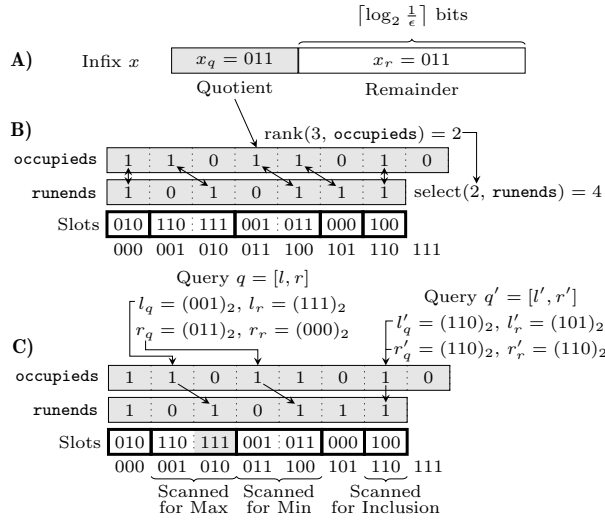


Figure 4: Diva splits an infix into a remainder and a quotient (Part A). It encodes the quotient in the `occupieds` bitmap while storing the remainder in contiguous runs in the array of slots, delineated using thick lines (Part B). Diva handles a query by locating the corresponding runs using rank and select operations, followed by scanning them (Part C). The remainders that satisfy the queries are highlighted.

Quotienting. An Infix Store applies Knuth’s quotienting technique [35] to its infixes, splitting them into *Quotients* and *Remainders*. Specifically, we take the $\lceil \log_2 \frac{2}{\epsilon} \rceil$ least-significant bits of an infix x as its remainder x_r and the rest of it (i.e., the $\log_2 T$ higher-order bits) as its quotient x_q . Figure 4-A illustrates this split for an infix $x = (011011)_2$.

Storing Quotients. Splitting each infix allows an Infix Store to succinctly encode its quotient within a dense bitmap called the `occupieds` bitmap. The i -th bit in this bitmap is set to 1 if an infix with a quotient equal to i exists and is set to 0 otherwise. For example, in Figure 4-B, infix x causes the bit at offset 3 (or 011 in binary) of the `occupieds` bitmap to be set to 1. This bitmap needs as many bits as there are possible quotients, which varies based on the predecessor and successor of the Infix Store. This amount is at most T bits since each quotient is $\log_2 T$ bits long.

Storing Remainders. Since there are exactly $T - 1$ keys in-between two samples, an Infix Store allocates an array of $T - 1$ slots, each $\lceil \log_2 \frac{2}{\epsilon} \rceil$ bits wide, to accommodate their infixes’ remainders. It stores remainders of infixes sharing the same quotient in a set of contiguous slots called a *Run*. Runs are stored in increasing order of their quotients. Each run spans ≈ 1 slot in expectation since the infixes are almost uniformly distributed, as described at the end of Section 3.1.

An Infix Store employs a $(T - 1)$ -bit bitmap with one bit per slot, called the `runends` bitmap, to delimit runs in an Infix Store. The i -th bit of this bitmap is 1 if the i -th slot in the Infix Store is the last slot of a run and is 0 otherwise. Figure 4-B shows an example of runs, depicted as boxes, and the `runends` bitmap delimiting them. In total, the `occupieds` and `runends` bitmaps consume 2 bits per slot in the array, or equivalently, 2 bits per infix.

Matching Invariant. Each run in an Infix Store is associated with exactly one 1 in the `occupieds` bitmap and one 1 in the `runends` bitmap. Since runs are stored in increasing quotient order, a *Matching Invariant* holds: the run associ-

ated with the i -th 1 in the `occupieds` bitmap ends at the slot corresponding to the i -th 1 in the `runends` bitmap. Figure 4-B illustrates this invariant with bidirectional arrows.

Locating Runs. Diva locates an infix’s run by first checking the bit corresponding to its quotient in the `occupieds` bitmap. If it is 0, the infix could not have been inserted and thus does not have a run. Otherwise, Diva locates the end of the infix’s run by employing rank and select primitives [45]. Formally, $\text{rank}(i, B)$ provides the number of 1s before the i -th bit in a bitmap B , and $\text{select}(i, B)$ gives the position of the i -th 1 bit in B . Diva locates the last slot in an infix x ’s run by evaluating the expression $\text{select}(\text{rank}(x_q, \text{occupieds}), \text{runends})$. In other words, it determines the run number it must jump to by computing rank over the `occupieds` bitmap and uses the result to locate the corresponding `runends` bit by computing select. Figure 4-B shows an example of this derivation with $x_q = 3$. Diva utilizes specialized hardware instructions [45] to apply rank and select efficiently.

Once Diva identifies the end of a run, it scans the remainders within that run from right to left until hitting the preceding run or the beginning of the Infix Store. During this scan, it reconstructs the run’s original infixes by concatenating the run’s quotient with each scanned remainder.

Bulk Loading. Diva bulk loads a sorted set of keys in a single sequential pass by inserting every T -th key into its trie and placing the infixes into the Infix Stores from left to right. This is significantly faster than other range filters, which incur the random access and CPU cost of hashing.

3.3 Query Processing

Div a processes a range query by searching for the query’s endpoints in its trie. A range query intersecting at least one sample in the trie immediately returns a positive since that sample represents a key within the range. In contrast, a range query that falls between two adjacent samples searches the corresponding Infix Store and returns a positive if at least one overlapping key exists.

Infix Store Range Queries. An Infix Store processes a range query over infixes $q = [l, r]$ by finding the quotients and remainders of the endpoints and considering two cases:

If the range spans multiple quotients ($l_q < r_q$), Diva checks if any quotient strictly between the endpoint quotients exists using the `occupieds` bitmap. If there is such a quotient, Diva answers with a positive, as all of the infixes in its run are strictly in the query range. Otherwise, it checks if any remainder in the left endpoint’s run is larger than its remainder l_r , or if any remainder in the right endpoint’s run is smaller than its remainder r_r . If either condition holds, Diva reports a positive and a negative otherwise. Query q in Figure 4-C shows an example. Here, Diva first checks the range of bits $[l_q + 1, r_q - 1] = [2, 2]$ in the `occupieds` bitmap for ones. As there is no bit set to 1 in that range, Diva scans the remainders in l_q and r_q ’s runs and compares them to l_r and r_r . As the former run has a remainder equal to $l_r = (111)_2$, Diva returns a positive.

If the range spans a single quotient ($l_q = r_q$), Diva checks if a run corresponding to that quotient exists. If not, Diva returns a negative. Otherwise, it scans this run for a remainder between the remainders of the endpoints l_r and r_r . If found, it returns a positive and a negative otherwise. This scan is efficient, as the expected size of a run is 1-2 slots. Query q' in Figure 4-C is an example. Here,

$l'_q = r'_q = 6$ and the bit at position 6 of the **occupieds** bitmap is one. Thus, Diva searches the relevant run for a remainder between $l'_r = (101)_2$ and $r'_r = (110)_2$. Since $\text{rank}(6, \text{occupieds}) = 3$ and $\text{select}(3, \text{runends}) = 6$, Diva checks the slots from Slot 6 backward until it exhausts the run. It returns a negative, as it does not find a remainder in the desired range.

False positives occur when query and key infixes collide, which is most likely when queries follow the dataset’s distribution. Diva’s trie learns this distribution, enabling a low FPR in this setting and potentially a better FPR in others.

Point Queries. Point queries are equivalent to specialized range queries with equal endpoints. Thus, they are handled by searching the trie and the corresponding Infix Store using the second case of the range query algorithm above.

3.4 Dynamicity

In addition to the static variant of Diva described in previous sections, we provide an equally performant design that supports dynamicity in exchange for ≈ 1 BPK more memory. This variant of Diva employs four core techniques, some of which draw on our prior works on dynamic filters. To better illustrate these techniques, we have reorganized the presentation of an Infix Store’s metadata bitmaps and its array of slots in Figure 5.

1) Overprovisioning Infix Stores. Diva overprovisions slots within each Infix Store to absorb insertions, as depicted in Figure 5-A. It inserts a new infix by mapping it to the slot at the offset equal to its quotient. Doing so uniformly scatters the runs across the Infix Store while leaving space in-between them. Diva stores each infix’s run in its mapped slot if there is space. If not, it makes room for the new infix by shifting other runs to the right until hitting an empty slot (and shifting to the left if there is no room to the right), similarly to Robin-Hood Hashing [10, 45]. The amount of shifting is moderate, since empty slots are spread evenly across the Infix Store.

2) Stretching Infix Stores. Once an Infix Store is 95% full, Diva expands it by a small factor ($\approx 1.05\times$). It does this using a technique we developed for Zeno Filter [32] called “*Stretching*,” whereby we stretch the distance between adjacent runs using a linear function to create more empty slots between them and accommodate more insertions.¹ For example, stretching the Infix Store in Figure 5-A results in Figure 5-B, which has two more empty slots. Stretching ensures a space utilization of at least 90% overall, as opposed to the 50% space utilization of other dynamic filters or hash tables that expand by a factor of 2 [47, 45, 20, 19, 25, 12].

3) Sampling Keys and Splitting Infix Stores. After $\approx T$ insertions into an Infix Store (and multiple stretching operations), the Infix Store should “*split*” in two, with a new sample in-between added to the trie. Splitting keeps runs short and queries efficient. It also refines the trie’s approximation of the data’s distribution, particularly for the parts of the key space that become denser.

An invariant of our design is to only store full, untruncated keys within the trie to allow for refining the key space in-between samples indefinitely. This invariant requires that only new insertions into the filter can be picked as samples,

¹In an Infix Store with n slots, Diva maps an infix x with a quotient of x_q to the slot at offset $\lfloor x_q \cdot n / (r - l) \rfloor$, where l and r denote the value of the bits in the predecessor and successor corresponding to the quotient x_q .

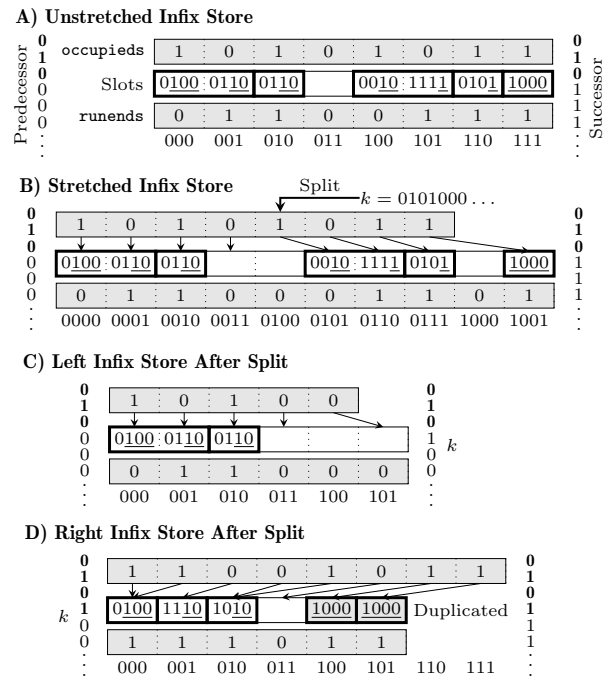


Figure 5: When an Infix Store runs out of overprovisioned space (Part A), Diva stretches it to make room for new insertions (Part B). The insertion of a new sample k into an Infix Store (Part B) splits it into a left (Part C) and a right (Part D) Infix Store. As a result of this split, old infixes may lose bits, elongating the unary paddings (underlined) and duplicating infixes with empty remainders. The predecessor and successor’s longest common prefix is highlighted in bold.

since all previously inserted keys have been truncated. Yet, this potentially leads to a vulnerability: deterministically inserting every T -th key as a sample into the trie would allow an adversary to create pathologically uneven splits and degrade performance. We address this risk by randomly picking each new insertion as a sample with a probability of $\frac{1}{T}$. This strategy ensures that, in expectation, every T -th key inserted into an Infix Store splits it and is taken as a sample. As such, the trie is still kept small ($\approx \frac{1}{T}$ of the data’s size). Moreover, since the samples collectively constitute a random $\frac{1}{T}$ fraction of the dataset, they lead to approximately even splits across the board, even though some splits may occasionally be uneven.

Figure 5-B illustrates an example of this procedure, which yields the left and right Infix Stores depicted in Figures 5-C and 5-D. The bit strings on the left and right of each Infix Store represent its predecessor and successor, respectively.

4) **Variable-Length Infixes.** Splitting an Infix Store causes the predecessor and successor of one of the new Infix Stores to have a longer common prefix. For example, the longest common prefix in Figure 5-D (shown in bold) is one bit longer than that of Figures 5-A and 5-B.

This longer common prefix takes over the higher-order bit(s) of the portion of each key previously taken as an infix, thereby shortening the original infix. To ensure each of these shorter infixes has an unambiguous quotient of the same length as before, Diva repurposes the higher-order bits of the infix's old remainder to become the lower-order bits of its quotient. Doing so splits each old run containing re-

mainders with different higher-order bits into shorter runs. Figures 5-B and 5-D present an example whereby the run previously in Slots 0101 and 0110 is split into two runs stored in Slots 000 and 001.

Despite old remainders becoming shorter, Diva still stores the infixes of new insertions with full-length remainders. This means that we truncate newer keys to a lesser extent as their region within the key space becomes denser. However, doing so necessitates storing variable-length remainders within an Infix Store. Following our work on InfiniFilter [20], Diva does this by padding each remainder to its original slot length with unary codes consisting of 0s and a delimiting 1. For example, the two-bit remainder in Slot 100 of Figure 5-A is stored with the two-bit padding 10, whereas the full-length remainder in Slot 101 is stored with a padding of 1. Due to the delimiting 1 bit, each slot must be one bit wider, adding a memory overhead of 1 BPK.

Each split reduces the length of the remainders within one Infix Store by one bit on average. Since each Infix Store is split after roughly T insertions (doubling its expected size), one can expect half of its remainders to be of full length, a quarter (from one split ago) to be shorter by one bit, an eighth (from two splits ago) to be shorter by two bits, etc. Taking a weighted average of the remainders' FPRs based on their lengths and proportion, we obtain an FPR that increases logarithmically with data size, in line with our past work on expandable filters. One can maintain a constant FPR by elongating new remainders, similarly to InfiniFilter's Widening Regime or Aleph Filters' Predictive Regime [20, 19, 44].

After many splits, some infixes' remainders may run out of bits, preventing Diva from computing an unambiguous quotient for them. Diva duplicates such infixes to account for all possible missing quotient bits and prevent false negatives, similarly to our work on Aleph Filter [19]. For example, it duplicates the right-most remainder in Figure 5-B to create the highlighted remainders in Figure 5-D in two different runs. Duplicating infixes complicates deletions, as there may be multiple duplicates to clean up. Our past work offers several methods for tackling this complexity [19].

Queries. Point and range queries are processed similarly to the static case, with the only difference being that the missing bits from shorter infixes are treated as wildcards.

4. EVALUATION

We evaluate Diva against existing range filters in the static setting in Section 4.1. We also conduct end-to-end experiments in the dynamic setting on top of WiredTiger [40], a popular B-Tree-based key-value store, in Section 4.2. We present further experiments in both static and dynamic settings in Sections 5.1 and 5.2 of our original paper [24].

Platform. We run experiments on a Fedora 39 machine with an Intel Xeon w7-2495X processor (4.8 GHz) with 24 cores and 48 hyper-threads. Our machine has an 80 KB L1 cache and a 2 MB L2 cache for each core, a 45 MB shared L3 cache, and 64 GB of main memory. It also has two SK Hynix 512 GB PC611 M.2 2280 80mm SSDs, which are used in the end-to-end experiments only.

4.1 Static Evaluation

Baselines. We compare Diva to all existing range filters except for bloomRF [41], as its implementation is closed-

source. As most other range filters only support integer keys, we specialize a version of Diva for integers to enable a fair comparison while still benchmarking the general-purpose version of Diva that supports variable-length keys. We implement both these versions in C++. All other baselines we compare to are implemented in C/C++ as well. We compile all filters using `gcc-13`.

Integer Datasets. Like prior work [59, 37, 54, 27, 41, 34, 52, 11, 18, 25, 12], we employ the following synthetic and real-world [38, 33] datasets for our evaluation over integers:

- **UNIFORM:** 200M uniformly sampled 64-bit integers.
- **NORMAL:** 200M 64-bit integer samples from $\mathcal{N}(2^{63}, 2^{50})$.
- **BOOKS:** 200M popularity scores for books on Amazon. This dataset is heavily skewed.
- **OSM:** 200M geocoordinates from the Open Street Map. It features several dense regions in the key space.

Query Workloads. For integer datasets, we sample a start key x and a length R to generate the range $[x, x + R - 1]$. We vary R to showcase the effect of range lengths on filter performance and FPR. For synthetic workloads, we choose the starting key x by sampling from the same distribution as the dataset. For real workloads, we sample x from the dataset and subsequently remove it from the set. We generate point queries by using $R = 1$.

Each of our workloads issue a total of 10M empty range queries, and we measure the FPR of each filter by dividing the number of positive results by the size of the query batch. We focus on filter CPU times in this section and measure end-to-end performance in the next section.

Experiment 1: Integer FPR vs Query Size Tradeoff. The top row of Figure 6 depicts the FPR of our baselines with a memory budget of 16 BPK, while the bottom row of Figure 6 compares their query latencies. Here, we vary query size on the x -axis. Figure 6 has both solid and dotted plots for Proteus. In the solid plots, Proteus is tuned using a sample of the queries, while in the dotted plots, it is tuned with range queries with uniformly distributed endpoints, simulating a workload shift. Such workload shifts do not impact the other range filters. We tune Rosetta and Memento filter to assume a maximum range query length of $R = 128$ to keep their FPR for short range queries low and their memory footprint in line with the other filters. If one keeps the memory footprint constant and tunes Rosetta for longer ranges, its query speed deteriorates. Tuning Memento filter for longer ranges yields faster queries at the expense of the FPR.

The FPRs of robust range filters, i.e., Rosetta, Grafite, and Memento filter, approach one as the range query length increases. As mentioned in Section 2, this is due to the filters issuing extra probes in their internal structures. This also increases the number of cache misses in Rosetta and Memento filter, slowing down range queries by orders of magnitude. Due to similar reasons, REncoder's FPR and speed also rapidly deteriorate with longer queries.

While SuRF approximately matches Diva's query performance, its FPR is at least an order of magnitude higher across experiments, as it maximally truncates the keys in its trie. Furthermore, SuRF is unable to curb its memory footprint under some datasets. For instance, in Figure 6, SuRF is missing from the BOOKS column since it requires at least 21 BPK to create a trie that differentiates all of the dataset's keys, exceeding the allotted memory budget.

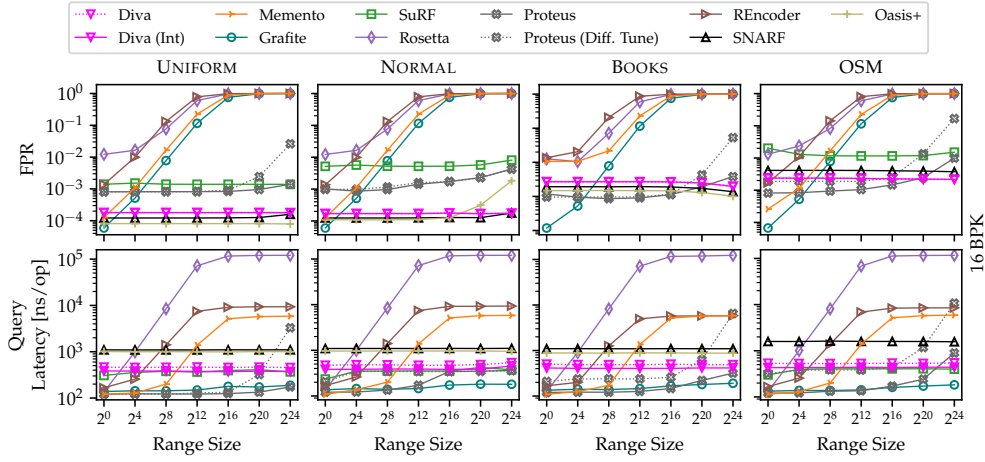


Figure 6: Diva provides the best balance between FPR and query latency for any workload across range query sizes.

SNARF and Oasis+ match Diva’s FPR but exhibit $\approx 2\times$ slower queries due to their slower binary searches. They also do not support variable-length keys and dynamic operations. Moreover, in some scenarios, such as in the OSM dataset, Diva achieves better FPRs than SNARF by a factor of $\approx 2\times$. This is due to floating point errors in SNARF’s distribution model. We exclude Oasis+ from the OSM column since its construction takes more than 12 hours.

Proteus’ FPR is slightly lower than that of Diva under the BOOKS and OSM datasets (due to some redundancy of the keys carrying over into Diva’s infixes) but is worse by orders of magnitude under the UNIFORM and NORMAL datasets. Moreover, Proteus’ dotted plots show that in the face of a workload shift, its FPR shoots up for medium to long queries, resulting in orders of magnitude higher FPRs than Diva. Although Proteus achieves faster queries than Diva in the absence of workload shifts, it becomes slower than Diva by an order of magnitude in the face of a workload shift as it performs more Bloom filter probes.

As shown in Figure 6, any filter that outperforms Diva in terms of FPR has significantly slower queries and vice-versa. Hence, Diva provides the best balance between FPR and query speed across the board for variable-length range queries. Moreover, as shown by the range size 2^0 , Diva is also competitive for point queries. As we show in subsequent experiments, Diva is also more general-purpose than its competitors since it achieves all other range filtering goals.

4.2 End-to-End Evaluation

Baselines. We compare Diva to Memento filter [25] by integrating both into WiredTiger [40], a popular B-Tree-based key-value store used to serve dynamic OLTP/HTAP workloads, to conduct an end-to-end evaluation. We only consider Memento filter as a baseline, as it is the only baseline with efficient dynamic operations. By default, the memory budget of each filter is set to 16 BPK.

Dataset and Workload. We use the BOOKS dataset with 200M keys. We load a randomly chosen $\frac{1}{64}$ fraction of this dataset into WiredTiger, construct the filters, and insert the remaining keys in a random order. We issue queries of length $R = 2^{10}$ with endpoints from the same distribution and collect measurements each time the dataset size doubles.

Experiment 2: End-to-End Query Latency. Figure 7 plots end-to-end range query latency measurements, with the x -

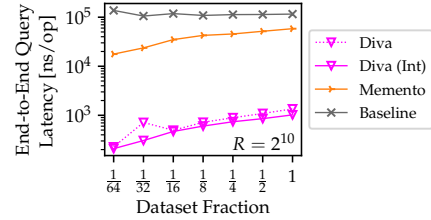


Figure 7: Diva achieves the best end-to-end query performance under dynamic workloads.

axis showing the fraction of the dataset ingested. Here, each key is associated with a 504-byte value, resulting in 512-byte key-value pairs on disk. We configure WiredTiger with a buffer pool that is 1% of the data size on disk. We trade some of this memory for the range filter to draw a fair comparison. The curve labeled “Baseline” in Figure 7 represents WiredTiger without a filter.

Div speeds up WiredTiger’s query processing by as much as three orders of magnitude. Moreover, since it supports variable-length queries without a deteriorating FPR, it beats Memento filter’s query latency by $\approx 85\times$.

5. FUTURE DIRECTIONS

Div opens up multiple fronts for future research. We are integrating Diva into OrcaDB, our lab’s fork of RocksDB [22]. Slow range queries have long been a fundamental challenge for LSM-trees [42, 22, 59, 37, 54, 27, 41, 34, 52, 11, 18, 25, 12], and Diva offers a general-purpose path forward for real-world datasets and workloads. Beyond range queries, Diva opens new research directions in storage engines, including adaptivity [39, 55, 12] (proactively mitigating recurring false positives) and efficient range deletes [53] (marking entire key ranges as deleted while preventing access to deleted keys in storage). One remaining limitation is Diva’s sensitivity to jagged data distributions, which commonly arise in natural language; however, we expect this issue can be addressed through entropy-encoding techniques [31, 58] over identical adjacent infixes.

6. ACKNOWLEDGEMENTS

This research was supported by NSERC grant #RGPIN-2023-03580.

7. REFERENCES

- [1] K. Alexiou, D. Kossmann, and P.-r. Larson. Adaptive range filters for cold data: avoiding trips to siberia. *Proc. VLDB Endow.*, 6(14):1714–1725, sep 2013.
- [2] Apache. Apache cassandra, 2024.
- [3] Apache. Apache druid, 2024.
- [4] P. Beame and F. E. Fich. Optimal bounds for the predecessor problem. In *Proceedings of the Thirty-First Annual ACM Symposium on Theory of Computing*, STOC ’99, page 295–304, New York, NY, USA, 1999. Association for Computing Machinery.
- [5] P. Beame and F. E. Fich. Optimal bounds for the predecessor problem and related problems. *J. Comput. Syst. Sci.*, 65(1):38–72, Aug. 2002.
- [6] M. A. Bender, M. Farach-Colton, R. Johnson, R. Kraner, B. C. Kuszmaul, D. Medjedovic, P. Montes, P. Shetty, R. P. Spillane, and E. Zadok. Don’t thrash: how to cache your hash on flash. *Proc. VLDB Endow.*, 5(11):1627–1637, July 2012.
- [7] I. O. Bercea, J. B. T. Houen, and R. Pagh. Daisy bloom filters. In H. L. Bodlaender, editor, *19th Scandinavian Symposium and Workshops on Algorithm Theory (SWAT 2024)*, volume 294 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9:1–9:19, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.
- [8] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, jul 1970.
- [9] A. Broder and M. Mitzenmacher. Survey: Network applications of bloom filters: A survey. *Internet Mathematics*, 1, 11 2003.
- [10] P. Celis, P.-A. Larson, and J. I. Munro. Robin hood hashing. In *26th Annual Symposium on Foundations of Computer Science (sfcs 1985)*, pages 281–288, 1985.
- [11] G. Chen, Z. He, M. Li, and S. Luo. Oasis: An optimal disjoint segmented learned range filter. *Proc. VLDB Endow.*, 17(8):1911–1924, may 2024.
- [12] Y. Chesetti, N. Eslami, H. Zhang, N. Dayan, and P. Pandey. Aeris filter: A strongly and monotonically adaptive range filter. In *Proceedings of the 2026 International Conference on Management of Data*, SIGMOD ’26, page 1670–1684, New York, NY, USA, 2026. Association for Computing Machinery.
- [13] J. G. Cleary. Compact hash tables using bidirectional linear probing. *IEEE Transactions on Computers*, 100(9):828–834, 1984.
- [14] G. Cloud. Bigquery enterprise data warehouse, 2024.
- [15] D. Comer. Ubiquitous b-tree. *ACM Comput. Surv.*, 11(2):121–137, jun 1979.
- [16] A. Conway, A. Gupta, V. Chidambaran, M. Farach-Colton, R. Spillane, A. Tai, and R. Johnson. Splinterdb: closing the bandwidth gap for nvme key-value stores. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC’20, USA, 2020. USENIX Association.
- [17] 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 ’10, page 143–154, New York, NY, USA, 2010. Association for Computing Machinery.
- [18] M. Costa, P. Ferragina, and G. Vinciguerra. Grafite: Taming adversarial queries with optimal range filters. volume 2 of *SIGMOD ’24*, page 1–23, New York, NY, USA, Mar. 2024. Association for Computing Machinery.
- [19] N. Dayan, I. O. Bercea, and R. Pagh. Aleph filter: To infinity in constant time. *Proc. VLDB Endow.*, 17(11):3644–3656, July 2024.
- [20] N. Dayan, I. O. Bercea, P. Reviriego, and R. Pagh. Infinifilter: Expanding filters to infinity and beyond. In *Proceedings of the 2023 International Conference on Management of Data*, volume 1 of *SIGMOD ’23*, New York, NY, USA, jun 2023. Association for Computing Machinery.
- [21] N. Dayan and M. Twitto. Chucky: A succinct cuckoo filter for lsm-tree. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD ’21, page 365–378, New York, NY, USA, 2021. Association for Computing Machinery.
- [22] S. Dong, A. Kryczka, Y. Jin, and M. Stumm. Rocksdb: Evolution of development priorities in a key-value store serving large-scale applications. *ACM Trans. Storage*, 17(4), oct 2021.
- [23] P. Elias. Efficient storage and retrieval by content and address of static files. *J. ACM*, 21(2):246–260, apr 1974.
- [24] N. Eslami, I. O. Bercea, and N. Dayan. Diva: Dynamic range filter for var-length keys and queries. *Proc. VLDB Endow.*, 18(11):3923–3936, July 2025.
- [25] N. Eslami and N. Dayan. Memento filter: A fast, dynamic, and robust range filter. In *Proceedings of the 2025 International Conference on Management of Data*, SIGMOD ’25, page 1–27, New York, NY, USA, 2025. Association for Computing Machinery.
- [26] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher. Cuckoo filter: Practically better than bloom. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, CoNEXT ’14, page 75–88, New York, NY, USA, 2014. Association for Computing Machinery.
- [27] Z. Fan, B. Ye, Z. Wang, Z. Zhong, J. Guo, Y. Wu, H. Li, T. Yang, Y. Tu, Z. Liu, and B. Cui. Enabling space-time efficient range queries with rencoder. *The VLDB Journal*, Aug 2024.
- [28] R. Fano. *On the Number of Bits Required to Implement an Associative Memory*. Computation Structures Group Memo. MIT Project MAC Computer Structures Group, 1971.
- [29] M. Foundation. Mariadb server: The innovative open source database, 2024.
- [30] M. Goswami, A. Grønlund, K. G. Larsen, and R. Pagh. Approximate range emptiness in constant time and optimal space. In *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA ’15, page 769–775, USA, 2015. Society for Industrial and Applied Mathematics.
- [31] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.
- [32] H. M. Kim, N. Eslami, and N. Dayan. Zeno filter: To

- infinity in tiny steps. *To Appear*, 2026.
- [33] A. Kipf, R. Marcus, A. van Renen, M. Stoian, A. Kemper, T. Kraska, and T. Neumann. Sosl: A benchmark for learned indexes. *NeurIPS Workshop on Machine Learning for Systems*, 2019.
 - [34] E. R. Knorr, B. Lemaire, A. Lim, S. Luo, H. Zhang, S. Idreos, and M. Mitzenmacher. Proteus: A self-designing range filter. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, page 1670–1684, New York, NY, USA, 2022. Association for Computing Machinery.
 - [35] D. Knuth. *The Art Of Computer Programming, vol. 3: Sorting And Searching*. Addison-Wesley, 1973.
 - [36] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 489–504, New York, NY, USA, 2018. Association for Computing Machinery.
 - [37] S. Luo, S. Chatterjee, R. Ketsetsidis, N. Dayan, W. Qin, and S. Idreos. Rosetta: A robust space-time optimized range filter for key-value stores. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 2071–2086, New York, NY, USA, 2020. Association for Computing Machinery.
 - [38] R. Marcus, A. Kipf, A. van Renen, M. Stoian, S. Misra, A. Kemper, T. Neumann, and T. Kraska. Benchmarking learned indexes. *Proc. VLDB Endow.*, 14(1):1–13, 2020.
 - [39] M. Mitzenmacher, S. Pontarelli, and P. Reviriego. Adaptive cuckoo filters. *ACM J. Exp. Algorithmics*, 25, Mar. 2020.
 - [40] MongoDB. Wiredtiger storage engine, 2024.
 - [41] B. Mößner, C. Riegger, A. Bernhardt, and I. Petrov. bloomrf: On performing range-queries in bloom-filters with piecewise-monotone hash functions and prefix hashing, 2022.
 - [42] P. O’Neil, E. Cheng, D. Gawlick, and E. O’Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, Jun 1996.
 - [43] R. Pagh and F. F. Rodler. Cuckoo hashing. In F. M. auf der Heide, editor, *Algorithms — ESA 2001*, pages 121–133, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.
 - [44] R. Pagh, G. Segev, and U. Wieder. How to approximate a set without knowing its size in advance, 2013.
 - [45] P. Pandey, M. A. Bender, R. Johnson, and R. Patro. A general-purpose counting filter: Making every bit count. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, page 775–787, New York, NY, USA, 2017. Association for Computing Machinery.
 - [46] P. Pandey, A. Conway, J. Durie, M. A. Bender, M. Farach-Colton, and R. Johnson. Vector quotient filters: Overcoming the time/space trade-off in filter design. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 1386–1399, New York, NY, USA, 2021. Association for Computing Machinery.
 - [47] P. Pandey, M. Farach-Colton, N. Dayan, and H. Zhang. Beyond bloom: A tutorial on future feature-rich filters. In *Companion of the 2024 International Conference on Management of Data*, SIGMOD/PODS '24, page 636–644, New York, NY, USA, 2024. Association for Computing Machinery.
 - [48] F. Putze, P. Sanders, and J. Singler. Cache-, hash- and space-efficient bloom filters. In *Experimental Algorithms, 6th International Workshop, WEA 2007*, volume 4525 of *Lecture Notes in Computer Science*, pages 108–121. Springer, 2007.
 - [49] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill, Inc., USA, 3 edition, 2002.
 - [50] R. Sears, M. Callaghan, and E. Brewer. Rose: compressed, log-structured replication. *Proc. VLDB Endow.*, 1(1):526–537, aug 2008.
 - [51] Snowflake. The snowflake ai data cloud, 2024.
 - [52] K. Vaidya, S. Chatterjee, E. Knorr, M. Mitzenmacher, S. Idreos, and T. Kraska. Snarf: A learning-enhanced range filter. *Proc. VLDB Endow.*, 15(8):1632–1644, apr 2022.
 - [53] F. Wang, D. Mo, and S. Luo. Don’t forget range delete! enhancing lsm-based key-value stores with more compatible lookups and deletes, 2025.
 - [54] Z. Wang, Z. Zhong, J. Guo, Y. Wu, H. Li, T. Yang, Y. Tu, H. Zhang, and B. Cui. Rencoder: A space-time efficient range filter with local encoder. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*, pages 2036–2049, 2023.
 - [55] R. Wen, H. McCoy, D. Tench, G. Tagliavini, M. A. Bender, A. Conway, M. Farach-Colton, R. Johnson, and P. Pandey. Adaptive quotient filters. *Proc. ACM Manag. Data*, 2(4), Sept. 2024.
 - [56] D. E. Willard. Log-logarithmic worst-case range queries are possible in space $\theta(n)$. *Information Processing Letters*, 17(2):81–84, 1983.
 - [57] X. Wu, F. Ni, and S. Jiang. Wormhole: A fast ordered index for in-memory data management. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.
 - [58] J. M. Yohe. Algorithm 428: Hu-tucker minimum redundancy alphabetic coding method [z]. *Commun. ACM*, 15(5):360–362, May 1972.
 - [59] H. Zhang, H. Lim, V. Leis, D. G. Andersen, M. Kaminsky, K. Keeton, and A. Pavlo. Surf: Practical range query filtering with fast succinct tries. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 323–336, New York, NY, USA, 2018. Association for Computing Machinery.

Making encodings easier to adopt

Jialin Ding
Princeton University
jialind@princeton.edu

Modern enterprises collect, store, and analyze increasingly large volumes of data. This relentless explosion in data volumes has driven the need for more effective encoding algorithms for reducing data size. In particular, this paper looks at *lossless* encodings, which support perfect reconstruction of the original data via decoding. Lossless encodings are widely used in database systems because they are able to not only compress data sizes, thereby reducing the cost of storing data, but also to minimize the I/O overhead of reading data, since compressed data requires less I/O bandwidth to read from persistent storage. In modern cloud-native database systems, where data is stored durably on cloud object stores and a hot set of data is cached on a compute node's local storage, encodings also enable more data to be cached as part of the hot set.

The idea of encodings is not new, and many tried-and-true encodings have been used in database systems for decades, including run-length encoding, dictionary encoding, delta and frame-of-reference encoding, and bitmap indexes. However, recent years have seen a rapid succession of newly-proposed encoding algorithms, driven by several trends. First, the rise of neural networks and LLMs has made floating-point numbers increasingly important due to their use in model weights and embeddings, and many new encoding algorithms specifically target floating-point numbers. Second, the increased use of specialized data types for certain domain-specific applications, such as the ROOT storage format for particle physics data, motivates the development of specialized encoding algorithms. Third, modern applications care not only about minimizing the size of encoded data, but also about the latency and throughput of encoding and decoding data, and different encoding algorithms make different tradeoffs between these competing objectives.

However, despite the plethora of newly-proposed encodings, very few have been adopted into widely-used commercial database systems. Furthermore, encoding algorithms which make it into one database system might not make it into others, with the result that each database system supports a slightly different subset of encoding algorithms. The reason for this haphazard adoption is simple: encoding algorithms are complex, and adoption requires extensive engineering effort for implementation and testing of correctness, performance, and security. For the developers of many database systems, the effort of adopting new encoding algorithms is simply too high. The authors call this the “ $M \times N$ problem” – supporting M encoding algorithms in N database systems requires $M \times N$ effort.

The authors of this paper propose (and implement) a solution to the adoption problem. They present Anyblox, a layer of abstraction which bridges encoding algorithms and database systems by enabling “self-decoding” datasets. Their core idea is that the logic for decoding a piece of encoded data is stored with the encoded data itself. In particular, the decoding logic is implemented as lightweight WebAssembly bytecode, which can be

compiled and executed on most popular architectures. This way, database systems no longer need to implement their own decoding logic for each unique encoding. Instead, a database system only needs to be able to interpret and execute the decoding logic shipped with the encoded dataset. Anyblox transforms the $M \times N$ problem into a much more tractable $M + N$ problem, where each of the M encoding algorithms implements decoding logic once and each of the N database systems implements support for Anyblox once. This is similar to how LLVM works for bridging programming languages and CPU architectures, which the authors draw direct inspiration from, and also how ODBC/JDBC works for bridging applications and database systems.

Anyblox’s “implement once, use anywhere” philosophy improves the *portability* of encodings across different database systems, but it naturally raises concerns about *performance* (since certain performance optimizations are only possible with tight integration between the host database system and encoding) and *security* (since it may be difficult to guarantee that the decoding logic won’t corrupt system state). The authors address both concerns: WebAssembly uses sandboxing to provably ensure that decoding logic is isolated from the host database system, thereby ensuring security. Furthermore, WebAssembly bytecode is known to achieve near-native speeds on many hardware platforms, which limits the hit on performance. (Interestingly, these natural advantages of WebAssembly have been identified by related efforts as well [1].) The authors also make clever use of memory mapping for zero-copy decoding and thread-local caching, which makes decoding even more efficient. The authors demonstrate that Anyblox can be integrated into popular database systems like DuckDB with minimal engineering effort, and the resulting integration achieves minimal performance degradation compared to native encodings.

Anyblox is a significant contribution in that it addresses a major issue which is often understudied by the research community: not how to achieve better performance or lower storage cost, but rather how to make adoption of new techniques as easy and frictionless as possible for existing systems.

The authors are open about Anyblox’s current limitations, especially in terms of performance. For example, database systems often get a performance boost by evaluating predicates directly on encoded data, but Anyblox is built to fully decode data into its uncompressed form and therefore cannot take advantage of this performance boost. The challenge of continuing to close the performance gap while maintaining the portability advantages of Anyblox will be an exciting direction for future research.

1. REFERENCES

- [1] X. Zeng, R. Meng, M. Prammer, W. McKinney, J. M. Patel, A. Pavlo, and H. Zhang. F3: The open-source data file format for the future. *Proc. ACM Manag. Data*, 3(4), Sept. 2025.

AnyBlox: Let Your Data Read Itself

Mateusz Gienieccko
giem@in.tum.de

Maximilian Kuschewski
maximilian.kuschewski@tum.de

Thomas Neumann
neumann@in.tum.de

Viktor Leis
leis@in.tum.de

Jana Giceva
jana.giceva@in.tum.de

Technical University of Munich

ABSTRACT

Research advancements in storage formats continuously produce more efficient encodings and better compression rates. Despite this, new formats are not adopted due to high implementation cost, and existing formats cannot evolve because they need to maintain compatibility across systems. Can this problem be solved by introducing a new abstraction? We answer affirmatively with AnyBlox, a framework for reading arbitrary datasets using lightweight WebAssembly decoders bundled with the data. By decoupling decoders from both systems and file format specifications, AnyBlox allows transparent format evolution, instance-optimized encodings, and enables mainstream adoption of research advancements. It integrates seamlessly with modern systems like DuckDB, Spark, and Umbra, while delivering solid performance and security guarantees.

1. INTRODUCTION

While researchers continuously develop faster and more space-efficient encodings [40, 12, 3, 1], as well as novel techniques like correlation-driven compression [27, 31, 51, 69, 52], these find **no adoption** in practice. New storage formats fail to gain traction, while existing formats undergo **ossification**, as datasets continue to be produced using outdated specifications to maintain compatibility with existing readers [39]. In this paper we investigate a future-proof abstraction layer between data encodings and data systems.

Monolithic database systems of the past fully controlled their storage representation, hiding problems inherent to format evolution from the users. However, the era of traditional data silos is over. The database community turns towards modular systems based on open formats [41], the industry moves towards open table formats in data lakes [78], and data scientists want to freely move between platforms when analyzing their existing data in exotic formats (e.g., High Energy Physics [29], bioinformatics [11, 20, 37, 21]).

These developments caused OLAP systems to break away from physical data independence advocated by Codd [23] and instead directly interact with data in its storage format (see Figure 1). Issues of format interoperability are now painfully apparent, as adoption becomes too expensive for

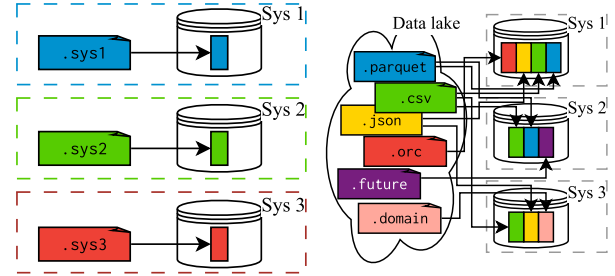


Figure 1: The $N \times M$ problem: In the past (left) every system controlled its data and internal format. Today (right) data is outside of systems' control. No system supports all formats and each system needs glue code for each format.

maintainers, preventing evolution and limiting users.

We identify the underlying issue as an instance of the $N \times M$ **problem**, where N systems having to support M formats leads to $N \times M$ implementation and maintenance effort. Any novel approach falls into a vicious cycle – an encoding needs to be popular enough to justify the cost of support, but it will not gain popularity without being widely supported. Similar problems were faced and solved before in different domains by introducing an abstraction layer, such as LLVM for compilers [44] and the Language Server Protocol for language tooling [54]. In this paper we ask:

What is the correct abstraction between modern data-processing systems and storage formats?

We analyze this question across four key dimensions that are desirable in a future-proof storage format abstraction: **(1) Portability:** how easy is it to integrate the solution across different systems and hardware architectures? **(2) Security:** what isolation guarantees does the solution provide? Can an error compromise the host system? **(3) Performance:** how does the solution fare in terms of query latency and throughput? **(4) Extensibility:** how easy is it to support a new data format with the solution?

In existing systems, supporting a new storage format usually means either adding a native decoder implementation, or running user-provided code through an extension mechanism. Native integrations can be fast, but they do not scale: each format must be implemented, tested, and maintained separately inside each host system, directly causing the $N \times M$ problem. In-process extensions improve development overhead, but are the worst case for security, since they amount to loading arbitrary code into the database process.

A natural alternative is to isolate extensions (containers/VMs, remote services, serverless UDFs) [61, 66, 9]. This improves isolation but shifts the bottleneck to data movement: decoder workloads are data-intensive, so the overhead of sending large intermediate results across process or network boundaries dominates in current approaches. Finally, approaches based on static verification are promising, but are currently unfeasible for complex decoders.

As Table 1 shows, state-of-the-art approaches tend to offer at most *two* of the four desired properties above at a time. This motivates our main contribution: **AnyBlox, a framework for self-decoding datasets**, which aims to simultaneously provide portability, security, performance, and extensibility. AnyBlox does this by bundling data together with WebAssembly bytecode decoders, allowing any consumer to read any data encoding without knowledge of its details.

Section 2 presents our **Bring-your-own-Decoder architecture** based around lightweight WebAssembly decoders, allowing arbitrary formats to be supported while providing solid performance and security guarantees.

Section 3 shows the integration of AnyBlox into a **wide range of database systems**, which employ vastly different data processing paradigms.

Section 4 showcases **multiple complex encodings** packaged into AnyBlox and evaluates their performance.

With these contributions we argue for a new paradigm in data ingestion – instead of systems decoding datasets based on format specification, **data should decode itself**, eliminating the unsustainable tight coupling between formats and systems. We outline the potential impact and future research directions in Section 5.

2. ANYBLOX

There are two main conceptual properties a future-proof data access solution must have: First, in order to solve the $N \times M$ problem and enable arbitrary data format evolution, it must abstract both data format internals away from systems, and system internals away from data decoders. Second, this abstraction layer must allow direct file access and data sharing use cases. These goals are at odds with each other, since direct access seemingly precludes any opportunity for introducing abstraction layers – where to put the format decoder that implements the abstraction? We argue that there is only one possible place for the decoder: alongside the data itself.

Self-decoding data solves the conceptual goals of abstraction with direct file access, but comes with its own set of challenges: How to ensure decoder portability across platforms? What are the security implications of running foreign code, and to make it performant? In the following, we

Table 1: Evaluation of different approaches according to our design dimensions. We analyze AnyBlox in Section 2.

Approach	Portability	Security	Performance	Extensibility
native	---	-	+++	---
extensions	-	---	+++	+/-
isolated ext.	+/-	+++	-	+
static verification	+	+/-	+/-	-
AnyBlox	+++	+	+	+++

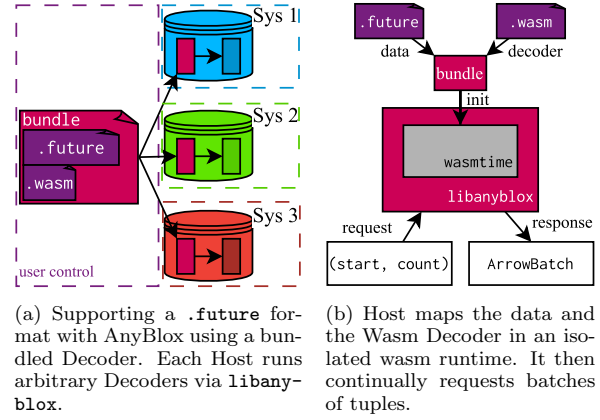


Figure 2: Integration of AnyBlox into Hosts.

describe our implementation of self-decoding data sets, *AnyBlox*, and how it solves these challenges. AnyBlox works on bundles of encoded data alongside **WebAssembly bytecode** defining a Decoder. The decoder can be packaged with the data in a single file, but a data lake might store decoder and data separately in an object store and tie them together via its metadata layer. Hosts can use the **AnyBlox Library (libanyblox)** to read any data following the API, which consists of the Decoder format (Section 2.1), the format of data returned to the Host (Section 2.2), and the metadata contained in the dataset (Section 2.3), as shown in Figure 2.

2.1 WebAssembly Decoders

WebAssembly (Wasm) is a specification of a virtual machine and its portable bytecode [30]. Its strength lies in a machine-verifiable guarantee on memory-safety and isolation from the host system [75]. It has been successfully utilized for isolation in different data processing applications, e.g. for ML workloads in Big Query [47], stateful serverless FaaS [63], and UDFs in databases [64], which makes WebAssembly a prime candidate for defining Decoders.

Portability. Wasm code is portable and can be compiled and run on a wide range of architectures, including browsers, mobile devices, embedded, and server workstations; as we show in Section 3, AnyBlox easily integrates into a variety of existing Hosts.

Security. WebAssembly’s specification is verified to uphold isolation guarantees [75]. Naturally, the guarantees depend on the *implementation* of the specification being correct. AnyBlox utilizes the Wasmtime implementation of Wasm and the Cranelift compiler to lower the Wasm bytecode to the native instruction set [16, 15], although AnyBlox does not rely on specific implementation details: both the compiler and the runtime are easily replaceable. Recent work suggests that the main source of vulnerabilities in WebAssembly is the compiler, namely the instruction selection and lowering process [73]. Formal verification of WebAssembly software isolation is an area of active research, in particular it was demonstrated that it is possible to implement a provably-safe Wasm sandbox with low performance overheads [13].

AnyBlox is built on top of open-source libraries and within the Rust ecosystem. The majority of the code is in safe Rust,

meaning it guarantees memory and thread safety. All the unsafe code is related to the Wasm memory maps and encapsulated in a few hundred lines of code in a single module, making it easily auditable.

Performance. In theory, Wasm allows for near-native performance, since it is JIT-compiled into the Host’s native instruction set. Moreover, AnyBlox avoids expensive data copying due to our memory manager design (see Section 2.6). While we devote Section 4 to performance evaluation, let us note here that research into closing the gap between Wasm and native code is both extensive and ongoing [67, 33]. In 2019 Jangda et al. identified instruction selection as the main source of performance degradation [32], and Yan et al. argue that common LLVM optimizations are ineffective when applied to WebAssembly [76]. Since then, major strides have been made in development of the novel Cranelift compiler. Crucially, AnyBlox can easily benefit from future improvements in Wasm compiler technology, as all the JIT and sandboxing details are encapsulated in `libanyblox` and not exposed in the public API.

Extensibility. Complex decoding schemes are easily implementable, as most general-purpose programming languages have a WebAssembly toolchain and compiler, including popular backends like LLVM [44], JVM [56], and CLI [25]. The main obstacle we have identified when porting decoding schemes to Wasm are SIMD instructions, which are a crucial component for some high performance compression schemes like PFOR [46], DELTA, dictionary, RLE [24], as well as data encodings like JSON [43]. WebAssembly exposes SIMD intrinsics, but developer effort is required to convert from x86/ARM SIMD to the different Wasm instructions. However, since WebAssembly defines SIMD intrinsics on the level of the bytecode, Wasm modules are fully portable: the JIT compiler selects the appropriate instructions for the Host machine.

2.2 Output Format

Since we want everyone to be able to write a single program that will decode their format, we need a clear definition of the decoding output. After considering natural requirements arising from our analysis of existing data processing systems and decoding schemes – columnar storage, support of most standard SQL types, low-overhead conversion to internal representations, portability and extensibility – we have chosen Apache Arrow [7].

Arrow is a robust standard, already supported as a data format by a number of database systems (e.g. DataFusion [41], LingoDB [?]), and fulfills all the above criteria. Arrow standardizes the most used data types for integers, floating-point numbers, decimals, date, time, strings, etc. Apache maintains high-quality libraries for efficient processing of Arrow in most commonly used programming languages. Arrow data is columnar, but complex multifield structures can be expressed as a logical type. Moreover, Arrow is extensible and allows custom data types.

2.3 Metadata

AnyBlox requires a thin metadata layer to tie the Decoder and data together. The Host needs to know the schema of the data that it loads from an AnyBlox file, usually during the query planning stage, ergo before decoding begins. We also require the **number of rows** in the compressed file, (an estimate of) the **size of decoded data**, and the **minimum rec-**

ommended batch size. These metrics aid the Host in query planning.

To make AnyBlox truly future-proof we also allow providing the **Decoder URI** and **Decoder cryptographic checksum**. This provides two distinct capabilities. First, the file may not contain the decoder and instead provide an external URL as the URI, allowing the Host to download it from a remote location. To maintain security, the downloaded payload should always be verified against the checksum. Second, once an encoding scheme becomes proliferated, a Host may decide to provide a more integrated native implementation of the Decoder. When opening a file it can compare its Decoder URI against a list of well-known URIs and instead decode the payload using its native Decoder.

2.4 Host Communication

AnyBlox is a modular system that needs to span the distance between an arbitrary data processing system and an execution environment for arbitrary WebAssembly code. We carve out two interfaces: between Host and `libanyblox`, and between `libanyblox` and an the WebAssembly Decoder. We present a top-level diagram of these API boundaries in Figure 2b.

To read a dataset the Host first initializes a **Decoder Job** (“open”), passing metadata, the dataset file, and the **WebAssembly Decoder**, which is a Wasm module exposing a `decode_batch` function (c.f., Section 2.5). Then, the Host can repeatedly request arbitrary tuple ranges, which are decoded into **Apache Arrow Record Batches**.

The architecture is illustrated in Figure 2b. A Host opens an AnyBlox bundle (data plus a Wasm Decoder) and then repeatedly requests tuple ranges. Each request invokes the Decoder’s `decode_batch` function and returns an Apache Arrow RecordBatch. This design allows Hosts to remain independent of the file format specification, while the decoder remains independent of any single Host.

To make repeated decoding calls efficient, AnyBlox supports a small *state* region that the Decoder can use for caching and for reusing allocations across calls. The state is an optimization only: decoders are required to be correct without it, which keeps the API simple and robust.

2.5 Decoder API

The Decoder needs to expose only one function with the following formal parameters:

- **i32 data**: the pointer to the place in Wasm linear memory where the encoded data starts;
- **i32 data_length**: the size of the encoded data;
- **i32 start_tuple**: the ID of the first tuple to decode;
- **i32 tuple_count**: the number of tuples to decode;
- **i32 state**: the pointer to the place in Wasm’s linear memory where the job state is stored;
- **i64 proj_mask**: a bitmask of columns to decode.

The Decoder, constrained by the WebAssembly standard, has no access outside of the sandbox aside from the `memory.grow` call. It is prohibited from modifying the pages containing the encoded data. Wasmtime catches any runtime errors or out-of-bound accesses, terminates the Decoder, and returns to the runtime.

The state page is zero-initialized at the start. State is only an optimization, allowing Decoders to cache metadata applicable to the entire dataset or reuse dynamic memory allocations across calls. In particular, since the Decoder must

perform correctly at the first call, and the runtime may clear the state page before each call, Decoders are guaranteed to be *idempotent*: their behavior can only vary based on the input parameters to `decode_batch` and the data – they have no access to system calls, thus no sources of randomness, clocks, etc. Therefore, because AnyBlox zeroes the state page at the start of every job, it guarantees that decoding the same file twice using the same parameters will return equivalent results.

2.6 Runtime

During initialization the Decoder is JIT compiled and the resulting module cached, so the costly compiler optimization step is incurred only once per Decoder. The linear memory object for the Decoder instance is created. Finally, an object representing the thread-local job is created as a handle to this instance and handed back to the Host. The compilation and isolated execution of Wasm code is done by Wasmtime [16]. Our core technical contribution is custom linear memory management for Decoder instances that ensures data integrity and low overhead access to both the input and output data.

The WebAssembly standard imposes that memory accessible to the Wasm instance is linear and starts at `0x0`. The guest code can allocate memory by invoking a `grow(x)` function, where x is the request size in 64 KiB pages, while the return value is the pointer to the start of the newly allocated region of x pages. The implementation of `grow` is provided by the Host, and the standard does not impose a specific mechanism for memory management on the Host's side.

Wasmtime implements the specification using a well-known Software Fault Isolation technique for 32-bit pointers [26]: It maps 8 GiB of protected virtual memory, while allocation requests from the Wasm instance are served by modifying the protection to allow read-and-write access for the required number of pages. This ensures that any memory access expressible in Wasm code falls within the virtual allocation, as the addressable pointer and length are both limited by $2^{32} - 1$, and thus requires no runtime bounds-checks in the compiled code. The physical memory is handled by the underlying OS's paging mechanism.

The challenge here is providing the entire encoded dataset to the Wasm instance through this linear memory abstraction. A naive solution would reserve the required chunk and copy the dataset into it, which would incur a heavy initialization cost as a potentially large amount of data would be copied. One could also envision a paging abstraction, where the Host would expose a function to read a given page of the dataset while ensuring copying happens only once per page; this approach suffers from excessive complexity, as we would be essentially reimplementing a page mapping and page-miss handling mechanism, and it could not be made transparent for the Decoder – it would have to work on the page-sized chunk abstraction of the dataset, which, among other things, would make vectorized algorithms more complicated and less efficient.

Instead, we propose transparent *data hooks* and thread-local memory pools. The dataset is available as a file descriptor, which we memory map into the virtual linear memory. This takes few system calls and is fast even for large datasets. The Decoder transparently accesses the data the same as any other area of its linear memory. This approach is similar to how Faasm handles shared memory regions by

swapping memory maps [63], but AnyBlox can provide a simpler and more cache-friendly mechanism since threads share only read access to the data.

AnyBlox makes no difference between file descriptors pointing to physical files, shared memory regions, or in-memory buffers like those maintained by `memfd_create` [50]. The initial memory required for the module's code, stack, and static data is allocated at the beginning of the memory. This is usually a couple megabytes. The hook is then mapped into the first available page after the initial pages, with the state page allocated immediately after. The Decoder's allocation requests are served from the following pages, growing dynamically as required.

The key property of this design is that linear memory regions can be cached in the thread-local pool and reused. While the Host is likely to decode a dataset with multiple threads, it is unlikely it will have two jobs running on the same thread at the same time, since context switching is detrimental to performance; even less likely to have two jobs on the same thread on the same dataset at the same time. Assume we allocate a linear memory region for a Decoder, it finishes its job, and we want to decide if another instance can reuse the memory. It is possible if the following constraints are upheld: a) the initial memory pages are large enough; b) the already hooked dataset is the same; c) all additionally allocated memory is zeroed (this is required by the WebAssembly standard). In practice, the initial regions are small (around a megabyte). When creating an instance we know the initial requirement as well as the dataset used, so the cache policy is straightforward: If a fitting map exists, use it, clear out the allocated memory with `madvise`, and protect all pages after the state page with `PROT_NONE`; if no fitting map exists and we are below memory instance limit, create a new virtual allocation; otherwise, pick the least recently used existing map and reuse it by unhooking the existing dataset and reinitializing.

The system can only infer that the dataset is the same if it comes from exactly the same opened bundle object. Sharing a single bundle between threads is important for performance, and simple, since the data is read-only. Moreover, if a query plan contains the same bundle as a source multiple times, the Host would benefit from merging them as one.

If the file is not directly available, e.g. it resides remotely and cannot be mapped to a file descriptor, it must first be downloaded. An alternative solution would be to enable lazy loading by custom handling of a page fault, e.g. via signal handling or specific operating system extension points [49]. In this scenario, the Host would provide code that populates a given page when it is first accessed by the Decoder. The Decoder would access memory as usual with full transparency, but only data it actually needs to access would be populated by the Host dynamically. This would not interfere with our memory caching mechanisms, allowing the materialized data to be reused later. AnyBlox does not yet implement this technique, but it is of interest for future work.

2.7 Discussion

AnyBlox is built to be future-proof. Our design allows for evolution without breaking existing implementations and datasets. In particular, (1) **Improvements to Wasm compilers** can be directly harnessed with an update to `libanyblox`; since the Wasm *bytecode* remains the same, this is a seamless upgrade that preserves backwards-and-forwards

compatibility. (2) A **new decoder dialect** could also be introduced in a compatible manner as long as there exists a Wasm transpiler. For example, if a statically-verified DSL for Decoders was designed, a Decoder in the DSL could be distributed with an automatic equivalent Wasm version being generated alongside. Systems running an older version of AnyBlox would use the Wasm code as before, while new versions could opt into running the DSL. (3) AnyBlox supports **Optional metadata**, allowing system-specific metadata that serves as a hint for one system but not another. An example of this is *sizeInBytes*, describing the estimated size of a fully decoded dataset in memory, which can aid distributed systems (e.g. Spark) in their network traffic estimations.

We believe these properties make AnyBlox a suitable solution for the problem of data format ossification. However, the use of WebAssembly also introduces some drawbacks. First, WebAssembly by default has a 32-bit address space into which AnyBlox maps datasets, meaning each mapped file has a maximum size of 4 GiB. Wasm recently gained support for using 64-bit addresses [17], but the performance impact of enabling this option is not yet well studied. Most systems using e.g. Parquet, keep file sizes well below 4 GiB [70, 6, 65] to facilitate cheap updates, so we expect this restriction to have little impact in practice. Second, Wasm code is currently not as fast as native code and does not support wide (> 128bit) vector instructions. Section 4 studies the performance impact of this limitation.

3. INTEGRATION

To demonstrate the portability of our solution we have integrated them into four fundamentally different systems: **DataFusion** [41], **DuckDB** [59], **Umbra** [55], and **Spark** [77]. We posit that these four Hosts cover a wide-enough variety of paradigms to strongly suggest AnyBlox poses no significant hurdles when integrating into an arbitrary data-processing system. Differences include:

Core paradigms. Umbra is a compiling system, and DuckDB and DataFusion are vectorized systems – two very different paradigms [35]. Spark focuses on distributed workloads, and only generates code for expressions.

Parallelism. Umbra and DuckDB are both morsel-driven with a central executor [45], while Spark and DataFusion use Volcano-style parallelism with exchange operators [28].

Programming languages. Umbra and DuckDB are written in C++, DataFusion in Rust, while Spark is written in Java and Scala, running in a managed JVM environment.

Maturity. Spark is a mature platform for diverse workloads and provides a plugin system; DuckDB is a widely used in-process analytical database with a robust extension API; Umbra is a closed-source research database that focuses on performance and provides no dedicated extension points; DataFusion was published only recently in 2024.

In the following we highlight integration details of interest for developers seeking to bring AnyBlox into their system. In case of DuckDB, its flexible system allows us to define a custom AnyBlox operator and distribute it as a plug-and-play module, requiring no invasive changes to the main DuckDB codebase. DataFusion provides a similar extension point for a custom table source, and the Spark integration uses a `.jar` plugin. We expect any system supporting external extension modules to allow easy integration in this manner. Umbra is not open-source, so we integrated AnyBlox directly into the

codebase and only provide a precompiled binary.

3.1 Internal Data Representation

While the Arrow format is optimized for zero-copy data transfer between systems using it as its internal data format like DataFusion, not all hosts do. SQL-based systems rarely have standardized logical types, and each host may choose to perform the SQL-to-Arrow type mapping differently. Arrow’s representation of primitive types is natural, so types such as `Int32` or `Float64` require no conversion. `Date` types need to be adjusted to the system’s internal representation, e.g., Umbra requires translation into the Julian calendar. `Bool` types are bit-packed in Arrow and might require conversion into bytes. This results in a zero-copy, in-place type conversion in most cases, with booleans being an exception.

Beyond the representation of primitive types, systems use different internal data layouts. DataFusion uses Arrow as its native format [41], allowing truly zero-copy data sharing with AnyBlox. DuckDB and Spark, like many modern systems [71, 62], support reading data in the Arrow format, converting it to their internal data layout on the fly. Thus, there already is mainline code that maps Arrow types and translates arrays to the internal representations. Umbra requires emitting code to load the values from Arrow buffers into registers based on the underlying physical type.

String types require the most care. Arrow supports two layouts: `Utf8` and `Utf8View`. `Utf8` contains a length and a pointer to the character data stored in a separate buffer. This can be easily translated to the Spark string format (regular JVM `String` type). However, both DuckDB and Umbra store strings using the small-string optimization [55], requiring more complex logic. `Utf8View` is actually based on this representation [8] as values may or may not contain offsets in separate buffers, which need to be translated to native pointers. Crucially, while the `Utf8`-to-`Utf8View` translation cannot be performed entirely in-place, the actual string data buffers do not need copying. The transfer is truly zero-copy in cases where the host and the decoder use the `Utf8View` layout for all strings.

3.2 Language Interoperability

Umbra and DuckDB are written in C++, while the AnyBlox Runtime is written in Rust. This makes the integration simple, as `libanyblox` can easily provide C++ bindings into Rust code. Integration into any system using a similar systems-level programming language with support for a C FFI should be similar. With a Rust-native system like DataFusion the integration is straightforward. Spark is written in Scala and Java, and thus requires a Java Native Interface bridge to communicate with `libanyblox`. Calls to AnyBlox return FFI-safe arrays – Arrow specifies a C data interface that provides uniform FFI across platforms, and thus the Arrow implementation on the JVM allows loading `ArrowArray` payloads into JVM objects.

3.3 Concurrent Scans

All integrations follow a similar pattern to implement parallel scanning of data, where an AnyBlox operator represents a full scan job that then gets divided into tasks for parallel processing. AnyBlox knows the exact size of the table from its metadata, and can skip columns based on the projection mask provided by the Host.

Spark and DataFusion decide the parallelism during query

Table 2: Approximate complexity of integrating AnyBlox into each of the Hosts by a single programmer. Spark also required writing a JNI bridge (433 LoC), which is reusable for any other JVM Host.

System	Paradigm	Parallelism	LoC	Days
Umbra	Compiled	Morsel-driven	1305	10
DuckDB	Vectorized	Morsel-driven	586	3
Spark	Hybrid	Volcano-style	756	15
DataFusion	Vectorized	Volcano-style	461	2

planning, letting the AnyBlox scan operator statically partition the workload without shared global state. This partitioning logic is independent of the underlying format itself and could be extended to multiple nodes (e.g., with Hadoop). The morsel-driven parallelism used by DuckDB and Umbra involves a global executor, with threads dynamically fetching work items to process in their local AnyBlox job. Threads keep local state, and, as an optimization, they may request large batches with `decode_batch` and process them in smaller chunks, e.g., the DuckDB global vector size.

3.4 AnyBlox Is Not A File Format

In recent years there has been a proliferation of file formats designed to meet the needs of current and future data management systems [2, 42]. Of note is the F3 format, which incorporates a similar idea of portable Wasm decoders for its Encoding Units [79]. AnyBlox, in contrast, is explicitly *not* a file format – it is a framework that can be used when building file formats, database systems, and applications. In particular, unlike F3, AnyBlox imposes minimal requirements on the layout of the underlying data and can be used to decode existing datasets with no changes. This is exemplified by our ROOT integration, where we use completely unmodified `.root` files as input.

AnyBlox is also decoupled from statistics and metadata. It does not enforce any metadata granularity, allowing the Host system to keep statistics on a column level, encoding page level, or with some other system-specific index. In this way AnyBlox functionally decomposes the storage encoding from the search acceleration layer [58].

The key characteristic of AnyBlox is its transparent, zero-copy data sharing with the sandbox regardless of the data layout. This flexibility allows AnyBlox integration to be a path towards future-proofing existing data formats. As an example, we implemented an experimental integration of AnyBlox decoders into the Parquet file structure, with Decoder definitions given in place of dictionary pages. A similar approach would work for the Vortex format [68].

4. EVALUATION

While the security guarantees of AnyBlox follow from the WebAssembly sandbox and portability is proven by our system integration in Section 3, what remains is substantiating our extensibility claims and showing the effectiveness of Wasm and the employed memory architecture for performance-intensive workloads. For that purpose, we evaluate the AnyBlox integrations using varied decoders, OLAP workloads, and microbenchmarks.

Experiments are performed on a 32-core Intel Xeon Gold 6430 machine with 256 GB of DDR5 RAM. Unless stated otherwise, samples are collected after two warm-up runs,

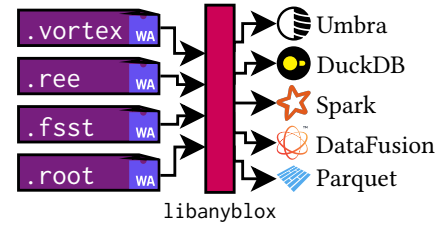


Figure 3: AnyBlox bridges the gap between a variety of storage encodings and arbitrary data processing systems.

which isolates the overhead of disk reads, Decoder compilation, and thread initialization. We present median results, but unaggregated results are available in our repository.

4.1 Did we solve the $N \times M$ problem?

Yes, AnyBlox solves the $N \times M$ problem, as we demonstrate in the following. We cover a wide range of formats by implementing (1) the encoding schemes FSST [12] and Run-End Encoding (REE), (2) Vortex [68], a novel columnar file format using state-of-the-art compression research [12, 3, 40, 1], and (3) CERN ROOT, a specialized binary format storing exabytes of particle physics data [18]. We complement this with an integration of AnyBlox into a variety of systems, done by a programmer with no prior experience with any Host codebase. No integration took more than two weeks, and all integrations were done in less than 2000 lines of code (LoC), as Table 2 shows. While neither the integration time nor the required lines of code are rigorous measurements of software complexity, they do indicate that AnyBlox can be straightforwardly ported to different systems. Figure 3 shows an overview. All integrations were straightforward to implement and required no further changes to any Host system. This enables great interoperability with minimal effort, allowing data transformations such as the following (native DuckDB syntax):

```
COPY (SELECT * FROM anyblox('./ParticleDecay.root')
      JOIN anyblox('./metrics.vortex') USING pType)
TO './ParticleDecayJoined.parquet'
```

4.1.1 Vortex

Vortex [68] is a novel format based on BtrBlocks [40] and other recent encoding research [3, 1]. Vortex outperforms existing alternatives using modern and multi-layered encoding schemes to achieve high compression rates, but suffers from the adoption problem outlined in Section 1. We built an AnyBlox decoder with little effort by compiling the Vortex codebase, specifying `wasm32` as the target for the Rust compiler, and excluding non-essential, OS-specific logic (such as file I/O), which cannot compile to Wasm. AnyBlox can bring this cutting-edge research codec into multiple database systems without any changes on the Host side, leading to immediate compression and performance improvements.

4.1.2 CERN ROOT

ROOT is a complex format developed at CERN since 1995 and is widely used across the High Energy Physics (HEP) community, with more than 2 EB of data stored in ROOT files [18]. Though much of this data is tabular, ROOT itself is not a relational format and allows encoding arbitrary C++

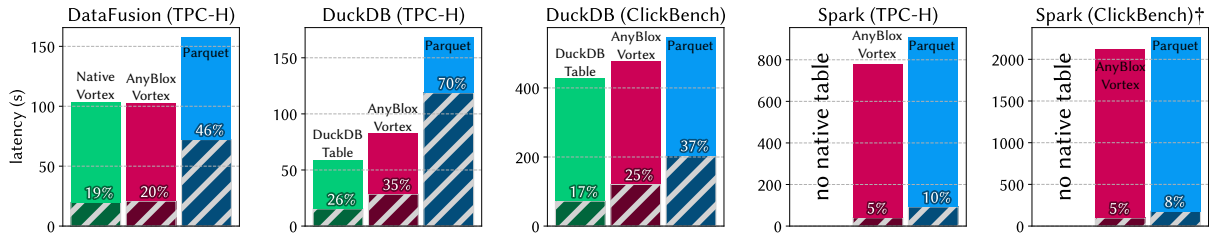


Figure 4: TPC-H and ClickBench evaluated on a single thread with DuckDB, Spark[†], and DataFusion. Time spent in the scan of the main table is shaded.

objects. Since ROOT is not supported in any conventional data systems, recent work on analyzing ROOT data with SQL had to convert all datasets to Parquet [29]. Our AnyBlox decoder (based on a Rust port [14] for a large subset of ROOT) instead makes ROOT files immediately readable by AnyBlox-enabled systems – no conversion required. Umbra surpasses the throughput of the native ROOT framework when performing an aggregation on particle decay data [48]. More importantly, Umbra’s parallel processing automatically works for ROOT files, yielding 20× better throughput on 32 threads, while the native framework offers no easy mechanism for parallelism.

4.1.3 FSST & Run-end encoding

FSST [12] and REE are lightweight encoding schemes that can be easily implemented in any language that compiles to WebAssembly. Both are also implemented in Vortex [68]. We use standalone Rust implementations in additional decompression microbenchmarks (omitted in this Research Highlight version).

4.2 Vortex Native vs. Vortex in AnyBlox

We show that (1) AnyBlox can be used to provide both compression *and* performance improvements by introducing novel encoding schemes; and (2) AnyBlox’s sandboxing overheads are negligible for complex decoding schemes as compared to a native solution. To that end, we evaluate the TPC-H benchmark with scale factor 20 in DataFusion and varying the representation of the largest table (`lineitem`). DataFusion is written in Rust, allowing direct integration of AnyBlox and Vortex. We use the native DataFusion Parquet reader on a file with the default compression scheme based on Snappy, which provides the best decompression throughput; the Vortex file format decoder inside AnyBlox; and the exact same decoder but compiled natively without any sandboxing. Predicate pushdown has been disabled.

Across all queries both Vortex implementations outperform Parquet with over 2× lower scan latency (c.f., Figure 4, left), and it provides a better compression rate (c.f., Figure 5). For Vortex, the bulk of processing time is spent executing relational operators (around 80%), whereas the Parquet reader spends almost half of its time decoding the file. The difference in performance between native and AnyBlox-based Vortex decoding is marginal, with a 5% increase in scan latency. However, this translates to a mere 1% increase in overall latency, which is within measurement noise.

[†]Spark only successfully executed 18 out of 24 selected ClickBench queries due to issues with the native Parquet reader unrelated to AnyBlox.

4.3 DuckDB Performance

Next, we test AnyBlox in a full database system – DuckDB. We compare AnyBlox Vortex to the native Parquet reader, but also include comparison against the native, non-portable table format of DuckDB. The performance results are shown in Figures 5, 4, and 6.

4.3.1 TPC-H

Across all queries the more efficient Vortex encoding outperforms Parquet despite the Wasm overhead, and AnyBlox is often close to the native baseline. In the single-threaded `lineitem` scan in Figure 4, both the native format and AnyBlox spend most of the time executing relational operators (74% and 65%, respectively), compared to Parquet, which has to spend most of the time (70%) decoding data. The biggest differences between in-memory table scans and AnyBlox occur in scan-dominated queries that select many columns from `lineitem` (Q1), high selectivity `lineitem` filtered scans (Q10, Q12 at 25% and 0.5% selectivity, respectively), and in Q21, where the triple self-join on `lineitem` exacerbates the cost, as Figure 6 shows.

4.3.2 ClickBench

TPC-H is an important standard benchmark, but fails to represent the real world, especially when it comes to string data [40, 27, 74]. We therefore turn to the ClickBench benchmark [22], which contains real-world web-traffic data in a single `hits` table. To focus on string compression, the following experiment extracts the five string columns that are heavily used in the majority of queries from this table, and compresses them using Parquet, Vortex, and DuckDB, which uses FSST [12] string compression in its native table format. Compared to DuckDB, Vortex achieves a 1.85× better compression rate due to its multistage encoding without relying on heavy-weight Snappy compression like Parquet, as Figure 5 shows.

Since AnyBlox can only support files at most 4 GB in length and compressed ClickBench exceeds that, we partition the data into 4 files with equal number of tuples in each. We do this for the native representation (split into 4 tables), Parquet, and Vortex alike. We restrict the workload to a subset of ClickBench queries focusing on string processing. In most queries, total processing time vastly outweighs scan times, Q29 being a prime example due to a complicated regular expression. In contrast, Q24 spends more time in the scan because DuckDB cannot push down the substring predicate into the non-native formats. Overall, AnyBlox spends only 25% of the total runtime in the scan.

In the multithreaded scenario, AnyBlox still outperforms native Parquet readers, as the following table shows (in mil-

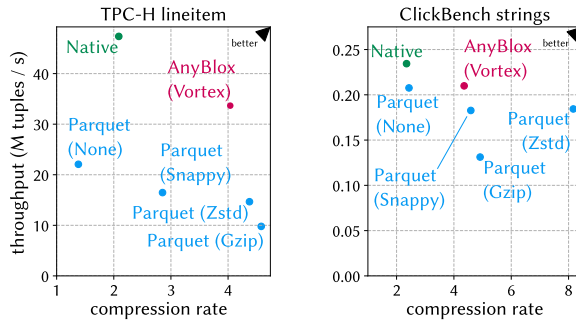


Figure 5: Compression ratio vs workload throughput on DuckDB for native tables, AnyBlox, and all Parquet compression codecs supported by DuckDB.

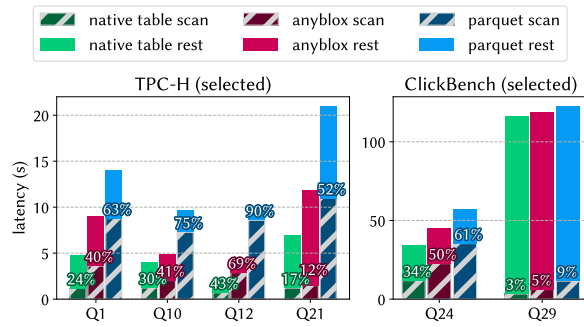


Figure 6: Selected queries from DuckDB TPC-H and ClickBench evaluation. Other systems exhibit similar behavior. Time spent in the scan of the main table is shaded.

lions of tuples per second, TPC-H scale factor 20 workload):

	Native table	AnyBlox (Vortex)	Parquet (Snappy)
1 thread	47.36 MT/s	33.67 MT/s	16.47 MT/s
32 threads	978.17 MT/s	593.18 MT/s	396.33 MT/s
ratio	20.65 ×	17.62 ×	24.06 ×

Parquet seems to scale marginally better, but at a large cost in terms of absolute performance [53].

4.4 Spark Performance

Finally, we evaluate TPC-H and ClickBench using Spark. Spark has no native data format, but has a native Parquet reader. Spark on a single thread is less efficient than DuckDB – most time ($\geq 90\%$) is spent executing operators. This means that, in Spark, scan performance has little impact on real-life OLAP workloads. For example, a 40% less efficient decoder would only make the Spark workflow 4% slower. Figure 4 shows this: Though Vortex in AnyBlox is more performant than Parquet, the overall impact is small. These characteristics are exacerbated in ClickBench, where the analytical operations dominate the scan even further.

4.5 Decoder microbenchmarks

To understand the performance of AnyBlox in detail, we perform microbenchmarks studying the impact of batch-size,

the scaling behavior of AnyBlox, and perform a breakdown of the per-thread and per-batch initialization cost. These microbenchmarks show that WebAssembly overheads depend on the decoder’s usage of SIMD, that AnyBlox scales well across threads, and that per-batch initialization cost is small. While lack of wide SIMD instructions in WebAssembly negatively impacts performance in microbenchmarks, our end-to-end results show that the overall impact is usually small because scans are only one component of query latency in real systems. The detailed results of these microbenchmarks are available in the full version of the paper.

5. SUMMARY AND OUTLOOK

We presented AnyBlox, a framework for self-decoding data that is easily **portable** across arbitrary data processing systems and **extensible** for arbitrary decoding schemes. Any system that implements an AnyBlox scan is able to **securely** and **performantly** read any format providing a WebAssembly decoder, solving the $N \times M$ problem and format ossification. Hosts can use bundled decoders directly, securely reference remote decoders using their cryptographic hashes, and transparently apply native decoders for popular encoding schemes. One can envision a future “Parquet v3” format that integrates AnyBlox, allowing arbitrary format evolution and instance-optimized encoding schemes.

Instance-optimized encoding schemes such as correlation-based compression [31, 52, 51], learned white-box compression [27], and entropy compression [60] can be very powerful, but their specialized nature makes their adoption even more difficult than more general-purpose schemes. AnyBlox paves the way for exploring a large space of clever, instance-optimized data representations – even pushing towards Kolmogorov complexity [38] – while still being future-proof and readable by any system.

Building future-proof data formats is but one of the problems that need to be solved to facilitate truly modular, future-proof, open database systems. The community has been calling for moving away from large, monolithic, overly complex systems for at least 25 years [19, 36, 57]. LLVM showed that an open framework can revolutionize the design of complex modern systems like compilers [44]. Recently, Lamb et al. presented an open and extensible query engine based around composable system design principles [41], proving such an approach can be successful. Execution plan frameworks have been made more extensible using sub-operators [10, 34], portable query optimizers [5, 4], and system-independent persistence mechanisms [72]. Our work joins this effort with an extensible and modular solution for arbitrary storage formats.

Acknowledgements

We thank Andrew Lamb for fruitful discussion and his feedback, and Maurice Scholtes for contributing the Parquet integration.

🇪🇺 Funded/Co-funded by the European Union (ERC, FDS, 101164556; ERC, CODAC, 101041375). Views and opinions expressed are however those of the authors only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

6. REFERENCES

- [1] A. Afrozeh and P. Boncz. The FastLanes Compression Layout: Decoding > 100 Billion Integers per Second with Scalar Code. *Proc. VLDB Endow.*, 16(9):2132–2144, May 2023. Publisher: VLDB Endowment.
- [2] A. Afrozeh and P. Boncz. The FastLanes File Format. *Proc. VLDB Endow.*, 18(11):4629–4643, July 2025.
- [3] A. Afrozeh, L. X. Kuffo, and P. A. Boncz. ALP: Adaptive Lossless floating-Point Compression. *Proc. ACM Manag. Data*, 1(4):230:1–230:26, 2023.
- [4] R. Alotaibi, Y. Tian, S. Grafberger, J. Camacho-Rodríguez, N. Bruno, B. Kroth, S. Matushevych, A. Agrawal, M. Behera, A. Gosalia, C. A. Galindo-Legaria, M. Joshi, M. Potocnik, B. Sezgin, X. Li, and C. Curino. Towards Query Optimizer as a Service (QOaaS) in a Unified LakeHouse Ecosystem: Can One QO Rule Them All? In *Conference on Innovative Data Systems Research*, Amsterdam, The Netherlands, 2024. www.cidrdb.org. arXiv: 2411.13704.
- [5] C. Anneser, N. Tatbul, D. Cohen, Z. Xu, P. Pandian, N. Laptev, and R. Marcus. AutoSteer: Learned Query Optimization for Any SQL Database. *Proc. VLDB Endow.*, 16(12):3515–3527, Aug. 2023. Publisher: VLDB Endowment.
- [6] Apache Parquet. Recommended parquet file size on hdfs, 2025.
- [7] Apache Software Foundation. Apache Arrow, 2023.
- [8] Apache Software Foundation. Apache Arrow Columnar Format - Variable-size Binary View Layout, 2023.
- [9] N. Armenatzoglou, S. Basu, N. Bhanoori, M. Cai, N. Chainani, K. Chinta, V. Govindaraju, T. J. Green, M. Gupta, S. Hillig, E. Hotinger, Y. Leshinsky, J. Liang, M. McCreedy, F. Nagel, I. Pandis, P. Parchas, R. Pathak, O. Polychroniou, F. Rahman, G. Saxena, G. Soundararajan, S. Subramanian, and D. Terry. Amazon Redshift Re-invented. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD ’22, pages 2205–2217, New York, NY, USA, 2022. Association for Computing Machinery. event-place: Philadelphia, PA, USA.
- [10] M. Bandle and J. Giceva. Database technology for the masses: sub-operators as first-class entities. *Proc. VLDB Endow.*, 14(11):2483–2490, July 2021. Publisher: VLDB Endowment.
- [11] A. Black, D. R. MacCannell, T. R. Sibley, and T. Bedford. Ten recommendations for supporting open pathogen genomic analysis in public health. *Nature Medicine*, 26(6):832–841, June 2020.
- [12] P. Boncz, T. Neumann, and V. Leis. FSST: fast random access string compression. *Proc. VLDB Endow.*, 13(12):2649–2661, July 2020. Publisher: VLDB Endowment.
- [13] J. Bosamiya, W. S. Lim, and B. Parno. Provably-Safe Multilingual Software Sandboxing using WebAssembly. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 1975–1992, Boston, MA, Aug. 2022. USENIX Association.
- [14] C. Bourjau. mALICE: An open source framework for analyzing ALICE’s Open Data, 2018.
- [15] Bytecode Alliance. Cranelift, 2016.
- [16] Bytecode Alliance. wasmtime, 2019.
- [17] Bytecode Alliance. Memory64 proposal for webassembly, 2025.
- [18] CERN. CERN ROOT Format. CERN, 1995.
- [19] S. Chaudhuri and G. Weikum. Rethinking Database System Architecture: Towards a Self-Tuning RISC-Style Database System. In *Proceedings of the 26th International Conference on Very Large Data Bases*, VLDB ’00, pages 1–10, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.
- [20] C. Chen, S. Nadeau, I. Topolsky, N. Beerenwinkel, and T. Stadler. Advancing genomic epidemiology by addressing the bioinformatics bottleneck: Challenges, design principles, and a Swiss example. *Epidemics*, 39:100576, June 2022. Place: Netherlands.
- [21] C. Chen, A. Taepper, F. Engelniederhammer, J. Kellerer, C. Roemer, and T. Stadler. LAPIs is a fast web API for massive open virus sequencing data. *BMC Bioinformatics*, 24(1):232, June 2023.
- [22] ClickHouse. ClickBench, 2022.
- [23] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, June 1970. Place: New York, NY, USA Publisher: Association for Computing Machinery.
- [24] P. Damme, D. Habich, J. Hildebrandt, and W. Lehner. Lightweight Data Compression Algorithms: An Experimental Survey (Experiments and Analyses). In *International Conference on Extending Database Technology*, pages 72–83, Venice, Italy, 2017. OpenProceedings.
- [25] ECMA International. Common language infrastructure (cli), 2012.
- [26] B. Ford and R. Cox. Vx32: lightweight user-level sandboxing on the x86. In *USENIX 2008 Annual Technical Conference*, ATC’08, pages 293–306, USA, 2008. USENIX Association. event-place: Boston, Massachusetts.
- [27] B. V. Ghita, D. G. Tomé, and P. A. Boncz. White-box Compression: Learning and Exploiting Compact Table Representations. In *Conference on Innovative Data Systems Research*, 2020.
- [28] G. Graefe. Encapsulation of parallelism in the Volcano query processing system. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, SIGMOD ’90, pages 102–111, New York, NY, USA, 1990. Association for Computing Machinery. event-place: Atlantic City, New Jersey, USA.
- [29] D. Graur, I. Müller, M. Proffitt, G. Fourny, G. T. Watts, and G. Alonso. Evaluating query languages and systems for high-energy physics data. *Proc. VLDB Endow.*, 15(2):154–168, Oct. 2021. Publisher: VLDB Endowment.
- [30] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien. Bringing the web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 185–200, New York, NY, USA, 2017. Association for Computing Machinery. event-place: Barcelona, Spain.

- [31] A. Ilkhechi, A. Crotty, A. Galakatos, Y. Mao, G. Fan, X. Shi, and U. Cetintemel. DeepSqueeze: Deep Semantic Compression for Tabular Data. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, pages 1733–1746, New York, NY, USA, 2020. Association for Computing Machinery. event-place: Portland, OR, USA.
- [32] A. Jangda, B. Powers, E. D. Berger, and A. Guha. Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 107–120, Renton, WA, July 2019. USENIX Association.
- [33] S. Jiang, R. Zeng, Z. Rao, J. Gu, Y. Zhou, and M. R. Lyu. Revealing Performance Issues in Server-Side WebAssembly Runtimes Via Differential Testing. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 661–672, 2023.
- [34] M. Jungmair and J. Giceva. Declarative Sub-Operators for Universal Data Processing. *Proc. VLDB Endow.*, 16(11):3461–3474, July 2023. Publisher: VLDB Endowment.
- [35] T. Kersten, V. Leis, A. Kemper, T. Neumann, A. Pavlo, and P. Boncz. Everything you always wanted to know about compiled and vectorized queries but were afraid to ask. *Proc. VLDB Endow.*, 11(13):2209–2222, Sept. 2018. Publisher: VLDB Endowment.
- [36] A. Khurana and J. L. Dem. The Modern Data Architecture: The Deconstructed Database. *login Usenix Mag.*, 43:36–40, 2018.
- [37] S. Knyazev, K. Chhugani, V. Sarwal, R. Ayyala, H. Singh, S. Karthikeyan, D. Deshpande, P. I. Baykal, Z. Comarova, A. Lu, Y. Porozov, T. I. Vasylyeva, J. O. Wertheim, B. T. Tierney, C. Y. Chiu, R. Sun, A. Wu, M. S. Abedalthagafi, V. M. Pak, S. H. Nagaraj, A. L. Smith, P. Skums, B. Pasaniuc, A. Komissarov, C. E. Mason, E. Bortz, P. Lemey, F. Kondrashov, N. Beerenwinkel, T. T.-Y. Lam, N. C. Wu, A. Zelikovsky, R. Knight, K. A. Crandall, and S. Mangul. Unlocking capacities of genomics for the COVID-19 response and future pandemics. *Nature Methods*, 19(4):374–380, Apr. 2022.
- [38] A. N. Kolmogorov. On tables of random numbers. *Sankhy: The Indian Journal of Statistics, Series A*, pages 369–376, 1963. Publisher: JSTOR.
- [39] L. Kuiper. Query Engines: Gatekeepers of the Parquet File Format, Jan. 2025.
- [40] M. Kuschewski, D. Sauerwein, A. Alhomssi, and V. Leis. BtrBlocks: Efficient Columnar Compression for Data Lakes. *Proc. ACM Manag. Data*, 1(2):118:1–118:26, June 2023. Place: New York, NY, USA Publisher: Association for Computing Machinery.
- [41] A. Lamb, Y. Shen, D. Heres, J. Chakraborty, M. O. Kabak, L.-C. Hsieh, and C. Sun. Apache Arrow DataFusion: A Fast, Embeddable, Modular Analytic Query Engine. In *Companion of the 2024 International Conference on Management of Data*, SIGMOD/PODS '24, pages 5–17, New York, NY, USA, 2024. Association for Computing Machinery.
- event-place: Santiago AA, Chile.
- [42] Lance contributors. Lance, 2025.
- [43] G. Langdale and D. Lemire. Parsing gigabytes of JSON per second. *VLDB J.*, 28(6):941–960, 2019.
- [44] C. Lattner and V. Adve. LLVM: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86, San Jose, CA, USA, 2004. IEEE Computer Society.
- [45] V. Leis, P. Boncz, A. Kemper, and T. Neumann. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 743–754, New York, NY, USA, 2014. Association for Computing Machinery. event-place: Snowbird, Utah, USA.
- [46] D. Lemire and L. Boytsov. Decoding billions of integers per second through vectorization. *Software: Practice and Experience*, 45(1):1–29, 2015. eprint: <https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.2203>.
- [47] J. Levandoski, G. Casto, M. Deng, R. Desai, P. Edara, T. Hottelier, A. Hormati, A. Johnson, J. Johnson, D. Kurzyniec, S. McVeety, P. Ramanathan, G. Saxena, V. Shanmugan, and Y. Volobuev. BigLake: BigQuery’s Evolution toward a Multi-Cloud Lakehouse. In *Companion of the 2024 International Conference on Management of Data*, SIGMOD/PODS '24, pages 334–346, New York, NY, USA, 2024. Association for Computing Machinery. event-place: <conf-loc>, <city>Santiago AA</city>, <country>Chile</country>, </conf-loc>.
- [48] LHCb collaboration (2017). Matter Antimatter Differences (B meson decays to three hadrons) - Data Files, 2011.
- [49] Linux Kernel Developers. userfaultfd(2) Linux User’s Manual, 2024.
- [50] Linux Kernel Developers. memfd_create(2) Linux User’s Manual, 2025.
- [51] H. Liu, M. Stoian, A. v. Renen, and A. Kipf. Corra: Correlation-Aware Column Compression. In *Proceedings of Workshops at the 50th International Conference on Very Large Data Bases, VLDB 2024, Guangzhou, China, August 26-30, 2024*. VLDB.org, 2024.
- [52] X. Lyu, A. Kipf, P. Pfeil, D. Horn, J. Giceva, and T. Kraska. CorBit: Leveraging Correlations for Compressing Bitmap Indexes. In *VLDB Workshops*, 2023.
- [53] F. McSherry, M. Isard, and D. G. Murray. Scalability! but at what cost? In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems*, HOTOS’15, page 14, USA, 2015. USENIX Association. event-place: Switzerland.
- [54] Microsoft. Microsoft LSP, 2016.
- [55] T. Neumann and M. J. Freitag. Umbra: A Disk-Based System with In-Memory Performance. In *Conference on Innovative Data Systems Research*, Amsterdam, The Netherlands, 2020. www.cidrdb.org.
- [56] Oracle. The java virtual machine specification, 2013.

- [57] P. Pedreira, O. Erling, K. Karanasos, S. Schneider, W. McKinney, S. R. Valluri, M. Zait, and J. Nadeau. The Composable Data Management System Manifesto. *Proc. VLDB Endow.*, 16(10):2679–2685, June 2023. Publisher: VLDB Endowment.
- [58] M. Prammer, X. Zeng, R. Meng, W. McKinney, H. Zhang, A. Pavlo, and J. M. Patel. Towards Functional Decomposition of Storage Formats. In *Conference on Innovative Data Systems Research*, Amsterdam, The Netherlands, 2025. www.cidrdb.org.
- [59] M. Raasveldt and H. Mühleisen. DuckDB: an Embeddable Analytical Database. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, pages 1981–1984, New York, NY, USA, 2019. Association for Computing Machinery. event-place: Amsterdam, Netherlands.
- [60] V. Raman and G. Swart. How to wring a table dry: entropy compression of relations and querying of compressed relations. In *Proceedings of the 32nd International Conference on Very Large Data Bases*, VLDB '06, pages 858–869. VLDB Endowment, 2006. Place: Seoul, Korea.
- [61] K. Saur, T. Mirmira, K. Karanasos, and J. Camacho-Rodríguez. Containerized execution of UDFs: an experimental evaluation. *Proc. VLDB Endow.*, 15(11):3158–3171, July 2022. Publisher: VLDB Endowment.
- [62] R. Schulze, T. Schreiber, I. Yatsishin, R. Dahimene, and A. Milovidov. ClickHouse - Lightning Fast Analytics for Everyone. *Proc. VLDB Endow.*, 17(12):3731–3744, Aug. 2024. Publisher: VLDB Endowment.
- [63] S. Shillaker and P. Pietzuch. Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 419–433. USENIX Association, July 2020.
- [64] M.-F. Sichert. *Efficient and Safe Integration of User-Defined Operators into Modern Database Systems*. PhD Thesis, Technische Universität München, 2024.
- [65] Snowflake. Introduction to extern tables in snowflake, 2025.
- [66] Snowflake, Inc. Introduction to external functions | Snowflake Documentation, 2025.
- [67] B. Spies and M. Mock. An Evaluation of WebAssembly in Non-Web Environments. In *2021 XLVII Latin American Computing Conference (CLEI)*, pages 1–10, 2021.
- [68] Spiral. vortex, 2024.
- [69] M. Stoian, A. v. Renen, J. Kobiolka, P.-L. Kuo, J. Grabocka, and A. Kipf. Lightweight Correlation-Aware Table Compression. *ArXiv*, abs/2410.14066, 2024.
- [70] The Apache Software Foundation. Parquet file size in apache impala, 2025.
- [71] TileDB, Inc. TileDB, Apr. 2024.
- [72] E. Tsalapatis, R. Hancock, R. Hossain, and A. J. Mashtizadeh. MemSnap $\dagger$ Checkpoints: A Data Single Level Store for Fearless Persistence. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS '24, pages 622–638, New York, NY, USA, 2024. Association for Computing Machinery. event-place: <conf-loc>, <city>La Jolla</city>, <state>CA</state>, <country>USA</country>, </conf-loc>.
- [73] A. VanHattum, M. Pardeshi, C. Fallin, A. Sampson, and F. Brown. Lightweight, Modular Verification for WebAssembly-to-Native Instruction Selection. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS '24, pages 231–248, New York, NY, USA, 2024. Association for Computing Machinery. event-place: <conf-loc>, <city>La Jolla</city>, <state>CA</state>, <country>USA</country>, </conf-loc>.
- [74] A. Vogelsgesang, M. Haubenschild, J. Finis, A. Kemper, V. Leis, T. Mühlbauer, T. Neumann, and M. Then. Get Real: How Benchmarks Fail to Represent the Real World. In *DBTest@SIGMOD*, pages 1:1–1:6. ACM, 2018.
- [75] C. Watt. Mechanising and verifying the WebAssembly specification. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2018, pages 53–65, New York, NY, USA, 2018. Association for Computing Machinery. event-place: Los Angeles, CA, USA.
- [76] Y. Yan, T. Tu, L. Zhao, Y. Zhou, and W. Wang. Understanding the performance of webassembly applications. In *Proceedings of the 21st ACM Internet Measurement Conference*, IMC '21, pages 533–549, New York, NY, USA, 2021. Association for Computing Machinery. event-place: Virtual Event.
- [77] 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*, NSDI'12, page 2, USA, 2012. USENIX Association. event-place: San Jose, CA.
- [78] M. A. Zaharia, A. Ghodsi, R. Xin, and M. Armbrust. Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics. In *Conference on Innovative Data Systems Research*, 2021.
- [79] X. Zeng, R. Meng, M. Prammer, W. McKinney, J. M. Patel, A. Pavlo, and H. Zhang. F3: The Open-Source Data File Format for the Future. *Proc. ACM Manag. Data*, 3(4):1–27, Sept. 2025.

Technical Perspective: PRISM and the Case for Strategic Opacity in UDF Optimization

Alkis Simitsis
Athena Research Center
Athens, Greece
alkis@athenarc.gr

The history of UDF optimization in relational databases reads as a story of increasing transparency [2]. Compilation makes UDFs faster but leaves them opaque to the optimizer. Batching and fusion amortize invocation costs but still treat UDFs as black boxes. Inlining, most notably demonstrated by Microsoft’s FROID system [3], aspires to the ultimate transparency by converting UDFs entirely into SQL, hence letting the optimizer reason about their contents. The trajectory seems inevitable: the path to better performance runs through making everything visible.

The core idea. PRISM, proposed by Arch et al. [1], upends this narrative. It shows that selectively restoring opacity, i.e., deliberately hiding parts of a UDF from the optimizer, yields better performance than full transparency. This is a counterintuitive result. Optimizers are designed to improve plans when given more information and yet PRISM shows that feeding them the full inlined translation of a complex UDF could harm performance. The LATERAL joins and deeply nested subqueries often produced by complete inlining overwhelm unnesting algorithms, prevent predicate pushdown, and generate bloated plans with unnecessary joins. PRISM reframes UDF optimization as a partitioning problem: rather than inlining an entire UDF into SQL, the framework decomposes each function, selectively inlines only the relational fragments helpful to the optimizer, and compiles the remaining procedural code into opaque native functions via *UDF outlining*, a technique borrowed in spirit from compiler engineering, where code is deliberately extracted into separate functions rather than inlined. This is a principled reversal of the prevailing wisdom: instead of asking “how do we inline everything?”, PRISM asks “what is the minimum we need to inline?”. The result is structurally simpler queries that are easier to unnest, enable data skipping, and avoid unnecessary joins.

The technique. The technical foundation of PRISM rests on a compiler-inspired decomposition of UDFs. The framework parses each function into a static single assignment (SSA) based intermediate representation (IR) organized as a tree of single-entry single-exit (SESE) regions: leaf, conditional, loop, and sequential. This structure is the linchpin of the approach. As every region has exactly one entry and one exit, any region can be extracted into its own function without restructuring the surrounding code. PRISM exploits this property through region-based outlining: it identifies the largest procedural regions that contain no embedded SELECT statements and extracts them into compiled, opaque functions. What remains for inlining is often just the relational fragments the optimizer can actually exploit. PRISM applies five complementary optimizations in sequence.

Query motion hoists embedded SELECT statements out of procedural control flow, separating relational code from procedural code so each can be optimized independently.

Region-based UDF outlining then identifies the largest procedural code SESE regions and extracts them into separate functions that are compiled to machine code, hiding them from the optimizer.

Instruction elimination leverages SSA’s single-assignment property to replace each variable use with its definition, progressively collapsing the UDF (frequently, to a single RETURN instruction) and eliminating all LATERAL joins.

Subquery elision bypasses inlining altogether by substituting the UDF’s return expression directly into the calling query, sidestepping the subquery wrapper that inlining would normally introduce.

Predicate hoisting injects UDF-derived filtering conditions into the query’s WHERE clause, hence unlocking data-skipping optimizations such as index scans and zone maps.

Together, these optimizations produce queries structurally simpler than what any single step could achieve alone. The power lies in their sequencing: query motion enables outlining, outlining enables instruction elimination, elimination enables elision, and hoisting addresses the orthogonal problem of predicate visibility.

The future. As SQL is increasingly extended to support emerging workloads such as LLM inference and semantic operators, effective UDF optimization becomes ever more critical. Yet many challenges remain. PRISM’s inline-versus-outline decisions are currently driven by structural heuristics (e.g., whether a region contains embedded queries) rather than by a cost model (e.g., [4]) that estimates downstream plan quality for alternative partitions. Developing such a model could capture cases where the structural rule is suboptimal, though at the expense of increased analysis complexity. Also, in engines lacking a mechanism to mark compiled UDFs as safe for parallel execution, outlined functions force single-threaded execution, partially eroding PRISM’s multi-threaded gains. PRISM also inherits a key limitation: loops with embedded queries still require full inlining, precluding finer-grained decomposition. More broadly, extending PRISM’s SSA-based analysis to dynamically typed UDF languages (e.g., Python, JavaScript), where type information is resolved at runtime and control flow is less structured, remains an open challenge. Similarly, understanding how the inline/outline partitioning interacts with unified compilation frameworks (e.g., MLIR, GraalVM) could yield even finer-grained decomposition strategies.

References

- [1] Samuel Arch, Yuchen Liu, Todd C. Mowry, Jignesh M. Patel, and Andrew Pavlo. 2024. The Key to Effective UDF Optimization: Before Inlining, First Perform Outlining. *PVLDB* 18, 1 (2024), 1–13.
- [2] Yannis Foufoulas and Alkis Simitsis. 2023. Efficient Execution of User-Defined Functions in SQL Queries. *PVLDB* 16, 12 (2023), 3874–3877.
- [3] Karthik Ramachandra, Kwanghyun Park, K. Venkatesh Emani, Alan Halverson, César A. Galindo-Legaria, and Conor Cunningham. 2017. Froid: Optimization of Imperative Programs in a Relational Database. *PVLDB* 11, 4 (2017), 432–444.
- [4] Johannes Wehrstein, Tiemo Bang, Roman Heinrich, and Carsten Binnig. 2025. GRACEFUL: A Learned Cost Estimator for UDFs. In *ICDE*. IEEE, 2450–2463.

Partial UDF Inlining

Samuel Arch¹, Yuchen Liu¹, Todd C. Mowry¹, Jignesh M. Patel¹, Andrew Pavlo¹

¹Carnegie Mellon University

sarch@cs.cmu.edu, yuchenl6@andrew.cmu.edu

tcm@cs.cmu.edu, jignesh@cmu.edu, pavlo@cs.cmu.edu

ABSTRACT

Although user-defined functions (UDFs) are a popular way to augment SQL’s declarative approach with procedural code, the mismatch between programming paradigms creates a fundamental optimization challenge. UDF inlining automatically removes all UDF calls by replacing them with equivalent SQL subqueries. Although inlining leaves queries entirely in SQL (resulting in large performance gains), we observe that inlining the *entire* UDF often leads to sub-optimal performance. A better approach is to analyze the UDF, deconstruct it into smaller pieces, and inline only the pieces that help query optimization. To achieve this, we propose UDF outlining, a technique to intentionally hide pieces of a UDF from the optimizer, resulting in simpler UDFs and significantly faster query plans. Our implementation (PRISM) demonstrates that UDF outlining improves performance over conventional inlining (on average 1.29× speedup for DuckDB and 298.73× for SQL Server) through a combination of more effective unnesting, improved data skipping, and by avoiding unnecessary joins.

1. INTRODUCTION

Application developers extend SQL’s capabilities by incorporating user-defined functions (UDFs) written in other programming languages (e.g., Python, JavaScript, PL/SQL) into their queries. However, queries with UDF calls are challenging for a database system’s query optimizer to reason about when choosing a good query plan because they are opaque functions written in a non-relational paradigm. As a result, queries with UDFs are often orders of magnitude slower than equivalent queries written without UDFs.

To address this impedance mismatch, researchers developed optimization techniques for UDFs, including *compilation* (i.e., generating specialized machine code for the UDF) [39, 37], *batching* (i.e., coalescing individual UDF invocations) [20, 15], and more recently, *UDF inlining* (i.e., translating a UDF into an equivalent SQL subquery) [45, 42, 22]. Inlining has shown the most potential of these techniques, providing up to 1000× performance improvements for workloads in a commercial DBMS [5].

The effectiveness of inlining stems from its ability to rep-

resent UDFs as SQL subqueries, a relational form that the DBMS can optimize. For example, $x=y$ becomes `SELECT y AS x`, and `IF/ELSE` blocks become `CASE WHEN` expressions. The DBMS then chains these translated expressions together using `LATERAL` joins, resulting in a SQL subquery equivalent to the original UDF. Inlining then replaces the original UDF call with the generated subquery that it injects into the calling query, leaving it in a purely relational form that enables the query optimizer to find better query plans.

The problem, however, is that inlining leaves many UDFs in an obfuscated form that the DBMS cannot reason about and optimize effectively. Figure 1 shows a UDF-centric query from Microsoft’s SQL ProcBench [19] (a benchmark modeled after Azure customer workloads). The figure demonstrates that inlining the *entirety* of the UDF generates complex subqueries containing `LATERAL` joins that are (1) challenging to unnest and optimize and (2) slow to execute.

A better approach is to deconstruct the UDF, identify the fragments beneficial for inlining, and expose them to the query optimizer. Now, UDF-centric queries become simpler and `LATERAL`-free by inlining only the code fragments necessary for better query plans. However, deconstructing a UDF is challenging for several reasons. First, users mix procedural and relational code by embedding `SELECT` statements inside control flow (conditionals/loops), preventing the DBMS from applying different optimization techniques to the same code block. Second, deciding which code to inline while minimizing code size is difficult. Lastly, users often write UDF predicates in `WHERE` clauses, which existing optimization techniques fail to exploit for data-skipping.

Given this, we present the **PRISM** UDF optimization framework. When an application registers a new UDF (i.e., with a `CREATE FUNCTION` command), PRISM performs analysis to carefully deconstruct the function into pieces, inlining some pieces and intentionally hiding others from the optimizer. The critical technique PRISM uses to achieve this is UDF *outlining*, which extracts UDF code fragments into separate functions that are intentionally not inlined, minimizing UDF code complexity. With the ability to either inline or outline pieces of a UDF, PRISM strategically restructures UDFs, maximizing the outlined pieces to reduce UDF complexity while inlining any pieces that may lead to better query plans. In addition, PRISM injects UDF predicates into a query’s `WHERE` clause to make data-skipping opportunities transparent to the optimizer.

To achieve this, PRISM performs five complementary UDF-centric optimizations that combine to address the drawbacks of inlining: (1) Predicate Hoisting, (2) Region-Based UDF

© Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment. This is a minor revision of the paper “The Key to Effective UDF Optimization: Before Inlining, First Perform Outlining”, published in PVLDB, Vol. 18, No. 1, ISSN 2150-8097. DOI: <https://doi.org/10.14778/3696435.3696436>

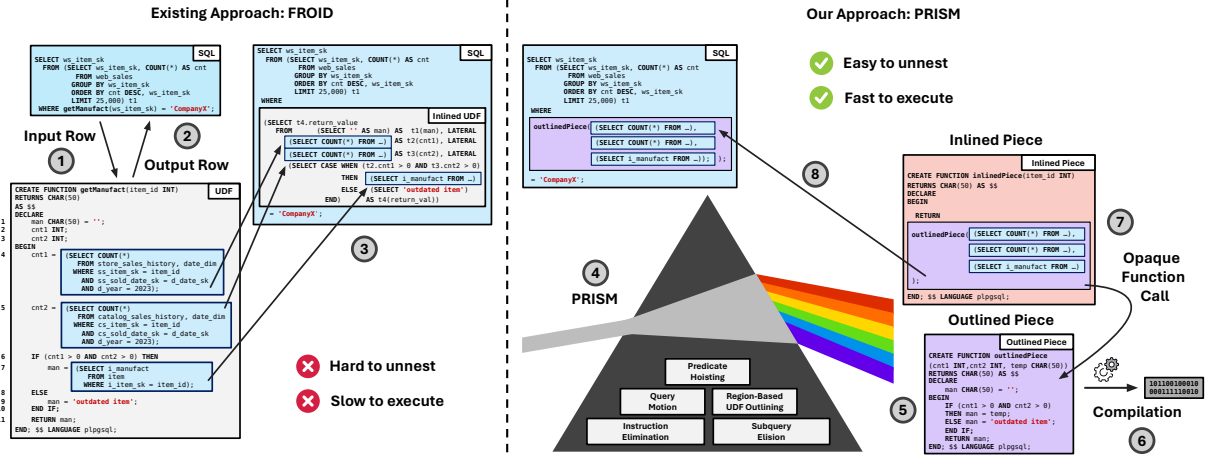


Figure 1: Overview of PRISM– The existing approach to UDF optimization (FROID) translates UDF execution from row-by-row oriented execution into a SQL subquery that it inlines into into the calling query. Our approach (PRISM) deconstructs a UDF into separate inlined and outlined UDF pieces. The framework injects inlined pieces into the calling query, but then compiles outlined pieces to machine code so that they are opaque to the DBMS’s optimizer.

Outlining, (3) Instruction Elimination, (4) Subquery Elision, and (5) Query Motion).

We integrated PRISM into DuckDB [41] and Microsoft SQL Server to evaluate our approach. Our experimental results show that after PRISM simplifies UDFs, they become straightforward for DBMSs to optimize and execute. As a result, PRISM accelerates UDF-centric queries from the Microsoft ProcBench [19] (by an average of $1.3\times$ on DuckDB and $298.7\times$ on SQL Server, and by a maximum speedup of $2270.2\times$ and $2997.9\times$, respectively).

We make the following contributions in this paper:

1. We identify the performance challenges introduced by UDF inlining during query optimization and execution: complex sub-queries containing LATERAL joins generated when inlining the entirety of a UDF (Section 2.4).
2. We introduce *UDF outlining*, a method to extract pieces of a UDF into separate functions to avoid inlining them, enabling more effective UDF optimization (Section 3.1).
3. We propose four complementary UDF-centric optimizations that restructure a UDF (Section 4), inlining only the pieces that lead to fast plans while maximizing the amount of outlined code and hoisting UDF predicates for data-skipping.

2. BACKGROUND

UDFs allow users to extend the database systems’ functionality by mixing imperative and declarative programming paradigms. But they create an impedance mismatch between UDFs and SQL that is challenging for both query optimization and execution. As we now describe, these problems have led to the development of methods to automatically optimize UDFs, but they all have drawbacks.

2.1 Compilation

Some DBMSs employ *compilation* techniques to reduce the engineering overheads associated with UDFs. As early as 2001, Oracle added support for “native compilation” of PL/SQL UDFs [39], transpiling them to C code, compiling the code into a shared library. This allows the DBMS to

execute a UDF as though it were a built-in function. Oracle also provides the PRAGMA UDF knob to allow the DBMS to reuse memory frames/data structures for function argument passing to reduce context switching overhead [12].

Microsoft SQL Server similarly added the ability to compile T-SQL UDFs to machine code in 2016 [37]. SingleStore added support to compile PL/pgSQL UDFs to LLVM IR in 2021 [46]. For dynamically-typed UDFs (i.e., Python UDFs), just-in-time (JIT) compilation is essential and is adopted by systems such as YeSQL [14, 13], BabelFish [17], Tuplex [48, 47], and Actian Vector [30]. However, compilation is ineffective if the chosen plan is sub-optimal.

2.2 Batching

Another earlier optimization technique is to *batch* invocations to amortize context-switching overheads of executing UDFs one row at a time. In 2008, researchers first proposed batching UDF invocations using program rewriting rules [20]. This approach only applies to DBMSs with a UDF interpreter rather than translating the batched queries to pure SQL. Later work refined UDF batching by rewriting UDF-centric queries to execute as a sequence of UPDATE statements on a temporary table [15]. Although this technique simplifies UDF-centric queries substantially, the overhead of materializing temporary tables is prohibitive.

2.3 Inlining

For a DBMS to achieve truly excellent performance for UDF-centric queries, the optimizer must reason about the UDF’s semantics as if it were written in SQL. UDF inlining [45] (developed in 2014) accomplishes this by translating UDFs into relational algebra (RA) that the DBMS can optimize effectively. Inlining translates each procedural instruction in a loop-free PL/SQL UDF to an equivalent RA expression. IF/ELSE blocks become CASE WHEN statements, assignments (i.e., $x = y$) become projections (i.e., SELECT y AS x). Then, the DBMS chains together the translated statements with APPLY operators¹, creating a single RA expression which is equivalent to the original UDF. Although the

¹T-SQL’s APPLY is similar to the SQL:1999’s LATERAL join.

2014 approach laid the foundation for future research, the main drawback is that it requires modifying the query optimizer to support extensions of the `APPLY` operator, which is not possible in closed-source systems and requires significant testing to avoid regressions.

Microsoft’s **FROID** pioneered a translation strategy to convert UDFs to relational algebra without requiring extensions to `APPLY` [42]. After inlining a UDF, the DBMS employs parallel, set-oriented execution plans instead of inefficient, serial, iterative plans. Microsoft released FROID with SQL Server 2019 and showed up to $1000\times$ performance improvements for customer workloads [5].

Although FROID only supports inlining of loop-free UDFs, further techniques have since lifted this restriction. Aggify [18] for example, uses dataflow analysis to replace all cursor loops inside a UDF with custom aggregate functions, enabling the UDF to then be inlined. However, both FROID and Aggify require changes to the DBMS internals.

More recently, the **Apfel** framework provides a pure SQL translation for UDFs [22]. For a given UDF, Apfel converts its procedural logic into a series of `SELECT` statements representing the same computation. The framework combines these statements into a single `SELECT` statement using `LATERAL` joins and then inlines the statement into the calling query. Apfel supports arbitrary control flow (including loops) via recursive common table expressions (CTEs). Any DBMS that supports `LATERAL` joins can execute inlined UDFs with Apfel, including systems without native UDF support.

2.4 Motivation

Inlining translates UDFs to complex subqueries containing `LATERAL` joins that are challenging for the DBMS to optimize and execute efficiently for the following three reasons:

(1) Difficult to Unnest: Inlining generates complex subqueries that are challenging for the DBMS to unnest (i.e., replace with join operators). Prior work shows that inlining UDFs produces subqueries that widely used DBMSs cannot unnest [15]. As a result, after inlining, the DBMS invokes the subquery once for each input row (similarly to how the original query invoked the UDF), resulting in extremely slow query execution. Although newer DBMSs like DuckDB unnest arbitrary queries [41, 38], existing systems do not perform arbitrary unnesting as it requires invasive changes to the query optimizer (using DAG-shaped rather than tree-shaped query plans, introducing special join operators). As we will show in Section 5, even for DuckDB, which supports arbitrary unnesting, these complex subqueries introduce additional join operators during the unnesting process, slowing down query performance.

(2) Prevent Data Skipping: Users often invoke UDFs as predicates in a query’s `WHERE` clause (see Figure 1). With inlining, however, the DBMS replaces UDFs with subqueries that it cannot push down into the leaves of the query plan. These scenarios prevent the DBMS from employing data-skipping optimizations (i.e., block skipping using zone maps or index scans), which causes queries to unnecessarily scan an entire table sequentially. Inlining obfuscates predicates, thereby blocking the optimizer from reasoning about them and identifying opportunities to avoid unnecessary work.

(3) Inefficient Query Execution: By translating a UDF into a sequence of `LATERAL` joins, inlining creates inefficient queries because DBMS optimizers struggle with join operators. Furthermore, inlining embeds these `LATERAL` joins in-

side a subquery, which, after unnesting, the DBMS replaces with a join, incurring additional overhead. Lastly, as we will show in Section 5, inlining translates loops into Recursive Common Table Expressions (CTEs) that often are an order of magnitude slower to execute than the loop in procedural form. Such issues argue for a UDF optimization technique that reduces the number of joins in a query.

3. PRISM OVERVIEW

PRISM is a UDF optimization framework that deconstructs a UDF into separate inlinable and outlinable pieces. Its goal is to inline the UDF pieces, exposing the code most amenable to SQL-style execution to the query optimizer while outlining as much of the remaining code as possible.

As shown in Figure 1, when the application installs a new UDF into the database (i.e., `CREATE FUNCTION`), PRISM examines the function’s contents to identify (1) which parts to inline into the calling query and (2) which parts to compile into machine code as separate functions through outlining. By only inlining a small portion instead of the entire UDF, PRISM eliminates all `LATERAL` joins. Thus, outlining provides two benefits: (1) it hides the outlined code from the optimizer through an opaque function, thereby minimizing the UDF complexity and resulting in simpler queries for the DBMS to optimize, and (2) the system can compile the outlined code to machine code to improve performance. As a result, PRISM’s optimizations overcome all the limitations described in Section 2.4.

We now describe PRISM’s architecture in more detail. We then discuss PRISM’s intermediate representation of UDFs to support our new optimizations in Sections 3.2 and 3.3.

3.1 Architecture

Figure 1 depicts the overall architecture of PRISM for the motivating example from Section 1. Compared to existing approaches which operate at query time (replacing the row-by-row UDF execution ① ② by inlining the UDF into the calling query ③), our approach (PRISM) occurs when a `CREATE FUNCTION` command is registered with the system. At this stage PRISM parses, binds, and translates the UDF to an intermediate representation (IR) (shown in Figure 2). PRISM then applies novel compiler optimizations ④.

First, PRISM decouples procedural and relational code with query motion (Section 4.1) to expose the largest possible code sequences for outlining. Next, UDF outlining (Section 4.2) extracts these code sequences into separate outlined pieces ⑤, which are then compiled to machine code ⑥, replacing the original code with opaque function calls ⑦ that minimize the UDF complexity. Then, instruction elimination (Section 4.3) removes as many instructions in the UDF as possible, collapsing a UDF down to a single `RETURN` instruction that does not require `LATERAL` joins when inlined ⑦. Subquery elision (Section 4.4) then replaces the original UDF call with the return value of the UDF ⑧, bypassing the inlining step and avoiding the generated subquery. Lastly, predicate hoisting analyzes loop-free UDFs and saves them as boolean predicates injected into a query’s `WHERE` clause, making data-skipping optimizations transparent to the optimizer. Unlike existing approaches that produce subqueries that are hard to unnest and slow to execute, PRISM produces `LATERAL`-free queries that are easy to unnest and faster.

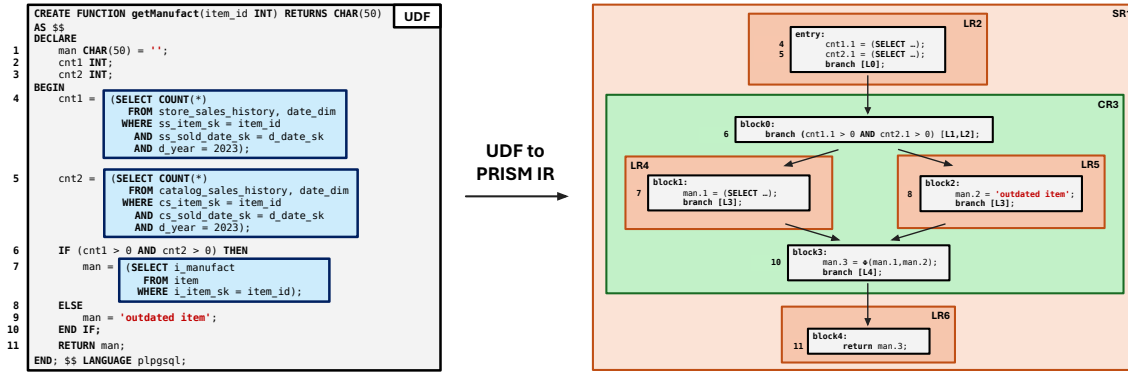


Figure 2: PRISM’s Intermediate Representation (IR) – PRISM uses an SSA-based IR, ensuring variables are assigned exactly once in the entire program. The IR also encodes high-level structured control flow (conditionals, loops) as a hierarchy of program regions. **LR** denotes a leaf region, **CR** denotes a conditional region, and **SR** denotes a sequential region.

3.2 Static Single Assignment (SSA) Form

PRISM represents UDFs as *control flow graphs* (CFGs) in *static single assignment* (SSA) form [43, 4] (shown in Figure 2). The CFG simplifies program analysis by breaking down high-level procedural constructs (loops, conditionals) into conditional and unconditional jumps (branches) between basic blocks [2]. Each basic block holds a sequence of non-branching instructions (assignments, function calls), followed by a single terminator instruction (a jump or return) that ends the basic block. In the CFG, nodes represent basic blocks and jumps create a directed edge from the source to the target block.

SSA form is an intermediate representation (IR) that ensures each variable is assigned exactly once. SSA form simplifies compiler design by making data flow explicit in the IR and is used in nearly all modern compiler implementations (e.g., LLVM) [34, 31, 49].

To convert to SSA form, existing variables (i.e., `man` from Figure 2) are assigned versions (i.e., `man.1`, `man.2`, `man.3`), one for each program point where the variable is assigned a value. At join points in the CFG (**block3**), ϕ -functions are inserted (i.e., `man.3 =  $\phi$ (man.1, man.2)`) where each ϕ argument indicates the variable’s value when control flow passes from a given predecessor. Note that ϕ functions are not executable, and the compiler eventually converts the program out of SSA form. Although PRISM uses SSA for simplicity, our optimizations also apply to non-SSA programs.

3.3 Regions

PRISM represents UDFs as a program structure tree [26], a hierarchy of single-entry single-exit (SESE) program *regions* [21, 1] (shown in Figure 2). Unlike the CFG that represents control flow as jumps, regions retain the high-level structured control flow of the original program (conditionals, loops) [26, 21, 1], simplifying certain compiler optimizations. There are four types of regions: leaf regions (a single basic block), conditional regions (blocks contained in an IF/ELSE), loop regions (blocks contained within loops), and sequential regions (a sequence of nested regions).

In Figure 2, the entire UDF is a single sequential region **SR1** comprised of nested subregions. The subregions are leaf region **LR2**, followed by conditional region **CR3**, and terminating with leaf region **LR6**. **CR3** contains block **block0**, which branches control flow between the IF and ELSE regions denoted by **LR4** and **LR5**, respectively, followed by

the fall through block **block3**, which joins control flow from the previous regions.

During SSA construction, ϕ functions, conditional branch instructions, and SELECT statements are placed in individual basic blocks ensuring that PRISM can cleanly construct program regions. By placing SELECT statements in leaf regions, non-SELECT code can be outlined independently (described in Section 4.2). By representing the program as a hierarchy of regions, each region has a single entry and exit point, enabling PRISM to outline regions as separate functions (shown in Section 4.2).

4. UDF-CENTRIC OPTIMIZATIONS

PRISM contains a suite of UDF-centric optimizations that restructure a UDF by splitting it apart into pieces and then optimizing them through a combination of UDF outlining and inlining. These optimizations are complementary and are meant to be combined together to achieve the best performance for UDF-centric queries.

For the examples in this section, we illustrate PRISM’s code changes to the UDF using (1) **blue** to represent SQL code (i.e., embedded SELECT statements, queries), (2) **red** to represent deleted code, and (3) **green** to represent new code.

4.1 Query Motion

Users often place SELECT statements inside of UDF control flow, such as conditionals and loops (including in our motivating example). However, mixing relational and procedural code introduces a challenge for UDF optimization. On the one hand, the DBMS should inline SELECT statements, making them transparent to the query optimizer. On the other hand, the DBMS should outline procedural control flow into opaque function calls, minimizing the UDF code size and the number of LATERAL joins that inlining generates. Coupling the relational code with procedural control flow prevents outlining of the procedural code, forcing the DBMS to inline the entire region and produce complex, hard-to-optimize subqueries.

A better approach is for the DBMS to *hoist* the SELECT statements outside of the procedural code (shown in Figure 3), splitting the UDF into pieces containing only procedural code (optimized for outlining) and relational code (optimized for inlining)². At the same time, it must carefully

²Although PRISM performs its optimizations on its inter-

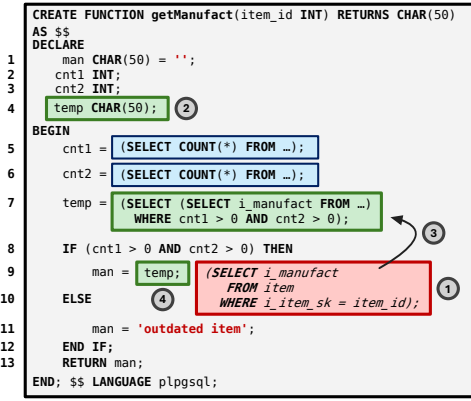


Figure 3: Query Motion – Hoisting SELECT statements outside of control flow (loops, conditionals) to expose larger code regions for outlining.

hoist the SELECT statement to avoid introducing redundant work and performance regressions.

Figure 3 illustrates the hoisting process for our motivating example. ① PRISM first chooses the SELECT statement on line 9 as a candidate for hoisting. Next, ② PRISM determines line 7 as the highest program point for hoisting and adds a temporary variable *temp* of the same type as the SELECT to the UDF. Then, ③ PRISM assigns the return value of the SELECT to the temporary, where the condition from line 8 now predicates the statement. Lastly, ④ PRISM rewrites the original assignment on line 9 to assign from the temporary instead of from the SELECT statement.

After query motion, PRISM decouples procedural and relational code, exposing larger regions of procedural code for UDF outlining. It is not always possible to hoist a SELECT statement outside of control flow (e.g., a loop that iteratively executes a SELECT that uses loop variables). In these cases, the DBMS falls back to inlining.

4.2 Region-Based UDF Outlining

With query motion, PRISM hoists SELECT statements above the procedural code, preparing the procedural code for UDF outlining. Although PRISM could inline the UDF at this stage, it is better to perform outlining for two reasons. First, outlining the procedural code into opaque function calls prevents the DBMS from inlining the functions, thereby reducing the number of LATERAL joins in the generated code and leading to more straightforward queries for the DBMS to optimize and execute. Second, after outlining, PRISM compiles the procedural code, making it an order of magnitude faster than if it were inlined. As a consequence, outlining as much procedural code as possible from the UDF is crucial to achieving high performance.

Outlining the maximum amount of procedural code into separate functions is non-trivial for several reasons. Query motion cannot hoist all SELECT statements, leaving some embedded inside procedural code blocks. Compiling these SELECT statements would hide them from the query optimizer, resulting in slow, iterative query plans [42], so PRISM leaves them for inlining. Yet, embedded SELECT statements (i.e., relational code in an IF/ELSE block) should only prevent outlining of the smallest surrounding code block (i.e.,

mediate representation, we show the transformations on PL/pgSQL UDFs in our figures.

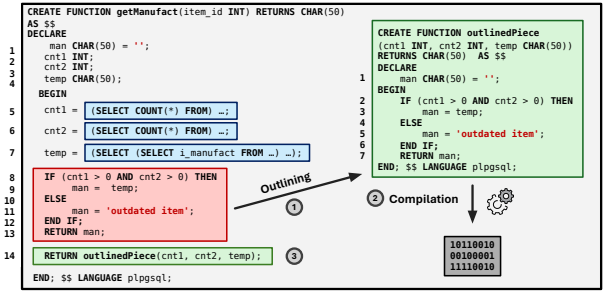


Figure 4: Region-Based UDF Outlining – Finding the UDF’s largest procedural code regions and extracting them into functions for compilation.

the IF/ELSE), not the other code blocks. Next, arbitrary UDF code may have multiple entry or exit points (multiple predecessor or successor blocks in the CFG), complicating code extraction. Lastly, PRISM should extract the largest possible procedural code at a time, minimizing the number of opaque function calls into and out of the UDF and exposing longer code sequences for compiler optimization.

The key insight necessary in addressing these challenges is to use a *region-based* approach to UDF outlining. As shown in Figure 2, PRISM represents UDFs as a program structure tree (a hierarchy of program regions) that represent structured control flow as a composition of four region types: (1) *sequential*, (2) *conditional*, (3) *loop*, and (4) *leaf*. Regions have precisely one entry and exit point [21, 1], ensuring that PRISM can extract any region into its own function. Further, PRISM handles embedded SELECT statements by never outlining a region if it contains another SELECT inside of it (including in its subregions).

Figure 4 illustrates the outlining process for our example UDF. First, ① PRISM identifies the largest region (lines 8–14) eligible for outlining. PRISM uses liveness analysis [1] to track variables entering the region (*cnt1*, *cnt2*, *temp*) and exiting the region (*man*). PRISM extracts this region into a separate function, with the entering variables becoming input arguments and the exiting variables becoming return values; PRISM creates user-defined types to handle multiple return values, similar to Aggify [18]. Next, ② the algorithm transpiles the extracted function to a C++ program, compiles it using *clang/gcc*, and then dynamically links it into the DBMS. Lastly, ③ the system removes the outlined region from the UDF and replaces it with a call to the compiled function. The extracted region becomes opaque to the query optimizer, resulting in queries that are simpler for the DBMS to optimize.

4.3 Instruction Elimination

Although region-based UDF outlining reduces UDF complexity, the resulting function still contains instructions that will produce a LATERAL join with inlining. Thus, PRISM must eliminate as many instructions as possible to produce simple, LATERAL-free queries that are straightforward to optimize. The challenge lies in how PRISM eliminates instructions while maintaining the UDF’s correctness and not introducing performance regressions.

We refer to each line of code in the UDF as an *instruction* and embedded queries in the UDF as *SELECT statements*. An instruction is eligible for elimination if it is *dead* (i.e., an instruction that does not affect the program’s result).

Therefore, PRISM must ensure that no other instructions in the program depend on it by tracking whether an instruction updates the state of a variable that is not used by any subsequent instructions. PRISM accomplishes this by taking advantage of the key property of SSA form: that each variable is assigned exactly once in the entire program. Therefore, to eliminate an instruction (i.e., $y = f(x)$) from a UDF, PRISM replaces every use of a variable with its definition (i.e., y with $f(x)$), eliminating the variable (i.e., y) from the program and making the defining instruction redundant.

When a variable has multiple uses, it introduces an important optimization decision. PRISM could duplicate the expression in the program, removing the instruction and reducing the number of LATERAL joins. Yet duplicating the expression may result in the DBMS evaluating the expression multiple times and performing worse than the original UDF. On the other hand, by not duplicating the expression, the UDF will still contain instructions that will result in complex subqueries with LATERAL joins after inlining.

PRISM addresses this trade-off by considering the number of uses and the cost of evaluating the expression. If a variable has one use, PRISM replaces it with the definition, saving a LATERAL join in the inlined UDF without causing regressions. However, when a variable has multiple uses, PRISM will treat it differently depending on whether the expression is a SELECT statement. If the expression is not a SELECT, then PRISM always duplicates it, relying on the DBMS's common-subexpression elimination (CSE) pass (available in both DuckDB [7] and SQL Server [42]) to identify duplicated expressions and evaluate them only once.

For SELECT statements, PRISM chooses whether to duplicate the expression depending on the target DBMS that will execute the UDF. For DBMSs that unnest arbitrary queries [38] (e.g., DuckDB), PRISM does not perform query duplication. But for DBMSs that only unnest LATERAL-free subqueries (e.g., SQL Server), PRISM applies query duplication, minimizing the UDF complexity as much as possible to enable unnesting. Only three out of 29 UDF-centric queries in our experimental analysis in require PRISM to decide whether to duplicate SELECT statements.

If an instruction defined by a SELECT has multiple uses and the DBMS unnests arbitrary queries [38], then PRISM does not eliminate it. Otherwise, the algorithm replaces every use of the instruction with its definition and removes the defining instruction from the UDF. It skips instructions that reference themselves (i.e., a cyclic dependency).

Figure 5 illustrates the instruction elimination process for the motivating example. First, PRISM replaces the uses of ① *cnt1* on lines 6–7 with its definition and then ② does the same for *cnt2*. Next, ③ PRISM replaces *temp* on line 7 with its definition on line 6. The last step ④ removes all dead variables from the UDF.

Through instruction elimination, PRISM collapses the entire UDF into a single RETURN instruction, resulting in a LATERAL-free subquery after inlining. Since DuckDB unnests arbitrary queries, PRISM skips code duplication in step ① since *cnt1* and *cnt2* have multiple uses. After instruction elimination, PRISM removes as many of the instructions in the UDF as possible, leaving the UDF in a much simpler form (in almost all cases, as a single RETURN instruction), which is easier for the DBMS to optimize and execute.

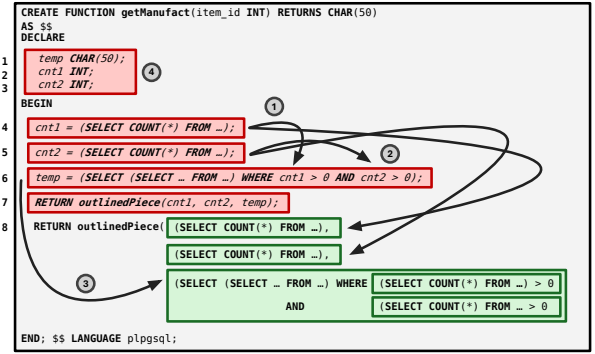


Figure 5: Instruction Elimination – Replacing each use of a variable with its definition, eliminating the instructions in a UDF, collapsing it down to a single RETURN instruction.

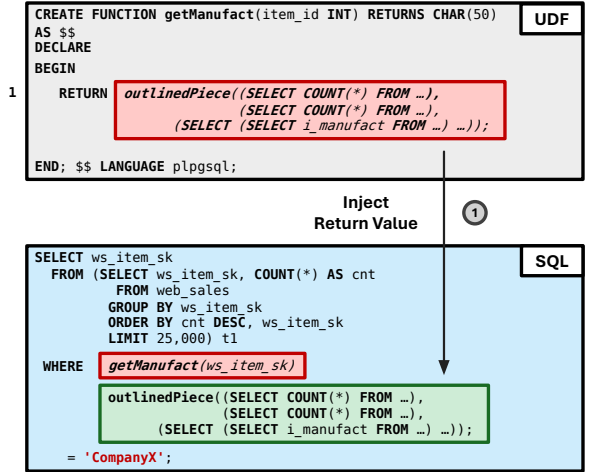


Figure 6: Subquery Elision – Replacing the UDF call with its return value, bypassing UDF inlining and the corresponding subquery.

4.4 Subquery Elision

Whenever possible, PRISM collapses UDFs into a single RETURN instruction to ensure LATERAL-free queries after inlining. However, inlining still wraps UDFs in subqueries that complicate query optimization and execution. To overcome this, PRISM performs *subquery elision* to replace the original UDF call with its return value, skipping inlining and avoiding the introduction of an unnecessary subquery.

Figure 6 illustrates subquery elision for our motivating example. Instead of inlining the UDF and generating a subquery (i.e., SELECT outlinedPiece(...)), ① the system directly substitutes the return value into the calling query. Although our experiments found that FROID already performs subquery elision whenever possible, to the best of our knowledge, we are the first to identify this optimization as a necessary step for effective UDF optimization.

5. EXPERIMENTAL ANALYSIS

We now present an evaluation of PRISM in DuckDB [41] (commit 53dc13d with a patch applied to support parallel CTEs [23]) and Microsoft SQL Server 2022. By integrating PRISM into DuckDB, the DBMS natively executes UDFs after registering them with the CREATE FUNCTION command.

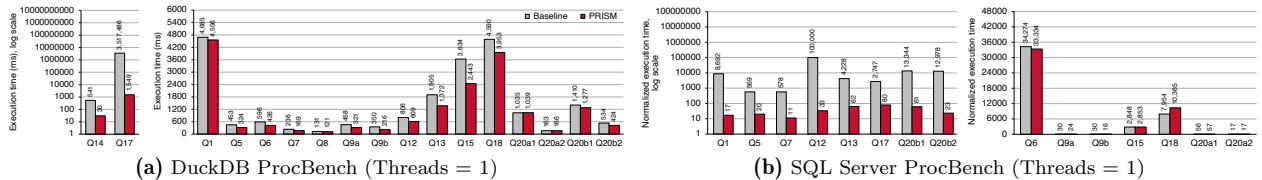


Figure 7: ProcBench (Single-Threaded) – Single-threaded execution times for DuckDB and SQL Server for the ProcBench queries with inlining (“Baseline”) and PRISM. Execution times are normalized for SQL Server (setting the maximum value to 100,000 units). Queries executing more than $10\times$ faster with PRISM are shown log-scale on the left-hand side, with the remaining queries shown on a linear scale.

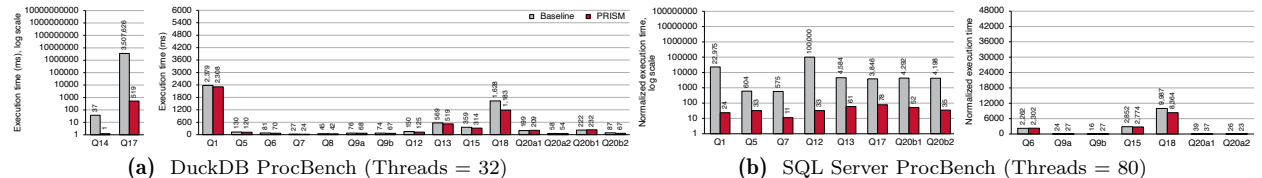


Figure 8: ProcBench (Multi-Threaded) – Multi-threaded execution times for DuckDB and SQL Server for the ProcBench queries with inlining (“Baseline”) and PRISM. DuckDB was executed with 32 threads to avoid a known bug with non-terminating queries [6], and SQL Server was run with the maximum degree of parallelism available ($\text{maxDOP}=80$). SQL Server results are displayed normalized, and queries with a $10\times$ speedup or greater are displayed log-scale.

To compare against inlining, we used Apfel [24] to generate the inlined UDF code for DuckDB. Since Apfel uses a different strategy than FROID, sometimes it generates slower queries than FROID. In these cases, we manually rewrite the queries to ensure a fair comparison. On SQL Server, we manually translate the optimized UDFs produced by PRISM from PL/pgSQL to Microsoft’s T-SQL syntax. We then use SQL Server’s “native compilation” feature to compile outlined UDFs to machine code [37]. We then inline the optimized UDF using SQL Server’s implementation of FROID [42, 5], rewriting the UDF to use a single `RETURN` instruction as required to ensure that the DBMS performs inlining.

We performed our evaluation on a machine with a dual-socket 20-core Intel Xeon Gold 5218R CPU (20 cores per CPU, $2\times$ HT), 192 GB DDR4 RAM, and a 960 GB NVMe SSD. We ran DuckDB on a single-socket with 32 threads to avoid a known bug with non-terminating queries [6] and use 80 threads for the maximum degree of parallelism (maxDOP) on SQL Server. We use the default index configuration for all workloads, and build additional column-store indexes [32] on every table on SQL Server. For each DBMS, we tune their configuration knobs to improve performance, pre-warm the buffer pool, and refresh statistics. We perform two warmup runs of each query and then five hot runs (with minimal observed variance), reporting the average execution time of the five runs.

5.1 Workloads (SQL ProcBench)

Microsoft released the SQL ProcBench in 2021 [19] as the first UDF-centric benchmark modeled after real-world UDFs on Azure SQL Server. ProcBench is based on the TPC-DS benchmark and contains 24 queries that invoke scalar UDFs. We use a scale factor of 10 (≈ 10 GB). We run 17 of the 24 queries, ignoring queries that use table-valued functions (TVFs) or UDFs invoked from stored procedures. We also skip queries Q8 and Q14 on SQL Server as these queries invoke UDFs with cursor loops. Aggify rewrites these cursor loops to custom aggregate functions that we could not compile on the Linux version of SQL Server 2022.

5.2 Single-Threaded Performance

We now evaluate DuckDB and SQL Server on single threaded execution with ProcBench. We run with a single thread to control for the effect of parallelism. We show in Section 5.3 that we observe similar results when running multi-threaded. ProcBench contains complex UDFs that inlining translates to subqueries with `LATERAL` joins. Therefore, ProcBench stresses PRISM’s ability to simplify UDFs and produce easy-to-unnest, fast-to-execute queries.

DuckDB: The results in Figure 7a show DuckDB’s performance improvement with PRISM. We use a log-scale axis for queries that are at least $10\times$ faster (i.e., Q14 and Q17), and a linear scale for the remaining queries. Q14 is $18\times$ faster with PRISM, as the UDF contains a loop that is faster to execute as a compiled outlined function compared to an inlined recursive CTE. Q17 executes $2270.8\times$ faster with PRISM than the inlined UDF because the latter introduces a slow cross-product join into the query plan during unnesting. The cross-product generates 250 billion tuples which enter the probe side of the hash join, taking over 56 minutes to finish. By comparison, PRISM optimizes the UDF, eliminating all `LATERAL` joins and subqueries, avoiding the cross-product and executing orders of magnitude faster (terminating in ≈ 1.5 s).

For the remaining queries, PRISM provides a speedup of $1.02\text{--}1.62\times$ (on average $1.29\times$). After inspecting the query plans with `EXPLAIN ANALYZE`, we find the extra runtime is due to executing additional join operators introduced when DuckDB unnests subqueries and `LATERAL` joins. By comparison, PRISM eliminates these subqueries and joins, resulting in faster query plans.

SQL Server: We next measure PRISM’s effect on SQL Server. We report the speedup of PRISM over the baseline in Figure 7b rather than reporting absolute numbers. We follow the same convention as above, reporting queries with over $10\times$ improvement using a log-scale and the remaining queries with a linear scale. Our first observation is that most ProcBench queries (eight out of 15) are at least an order of magnitude (on average $559.1\times$) faster with PRISM relative

Table 1: ProcBench Subquery Unnesting – For each system (FROID, PRISM), applied to each DBMS (SQL Server, DuckDB), the table indicates whether the DBMS unnests all subqueries in a given Bench query (i.e., replaces them with join operators).

Technique		ProcBench Queries															
		Q1	Q5	Q6	Q7	Q9a	Q9b	Q12	Q13	Q15	Q17	Q18	Q20a1	Q20a2	Q20b1	Q20b2	
SQL Server	Inlining	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	
	PRISM	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	
DuckDB	Inlining	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	
	PRISM	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	

to the baseline. The source of this improvement is that SQL Server cannot unnest the subqueries generated by inlining but unnests PRISM’s simplified UDFs. Through unnesting, the DBMS replaces inefficient, iterative subqueries with set-oriented join operators that provide an algorithmic advantage and faster query performance. Our analysis of query plans for the ProcBench (Table 1) shows that SQL Server only unnests four out of 15 of FROID’s inlined queries, compared to 12 with PRISM. Contrast this with DuckDB, which unnests all ProcBench queries [38, 40]. By simplifying PRISM’s generated UDFs and hiding as much code as possible through outlining, SQL Server unnests the simplified queries, resulting in orders of magnitude faster query plans. For the remaining queries, PRISM is faster by avoiding unnecessary LATERAL joins (on average 1.12× faster). The exception is Q18, which is slower with PRISM due to duplicating SELECT statements.

5.3 Multi-Threaded Performance

To evaluate whether PRISM’s benefits generalize to a multi-threaded setting, we perform the same measurements as in Section 5.2 but with the maximum number of threads. To avoid a known bug in DuckDB that prevents queries from terminating [6], we use a maximum of 32 threads in our DuckDB experiments.

DuckDB: Comparing single-threaded and multi-threaded performance (Figures 7a and 8a), PRISM provides similar improvements. We attribute DuckDB’s stable scalability to its execution of physical operators in both query plans with HyPer-style morsel-based parallelism and scheduling [35].

SQL Server: Unlike DuckDB, PRISM’s performance improvements are lower on SQL Server when executing queries multi-threaded. We attribute this gap to an implementation detail of SQL Server that forces single-threaded query plans when invoking non-inlined UDFs [42] (i.e., the outlined UDF pieces generated by PRISM). Hence, PRISM has a diminished benefit on SQL Server with multiple threads. SQL Server could address this problem by providing a PARALLEL SAFE annotation (similar to PostgreSQL), allowing outlined UDF pieces and the entire query plan to run in parallel.

6. RELATED WORK

We now discuss prior work related to our approach. We refer the reader to Section 2 for our overview of existing UDF optimization techniques (i.e., compilation, batching, and inlining).

Subquery Unnesting: [29] introduced the first technique to unnest subqueries. [44] extended their work by providing unnesting rules for more complex subqueries. SQL Server models subqueries using the APPLY operator, relying on rewrite rules to remove the APPLY operator from the query

plan [16, 8]. However, this approach cannot unnest arbitrary queries [38]. In the context of UDF inlining, [15] demonstrated that SQL Server fails to unnest inlined UDFs when the generated subquery contains nested APPLY/LATERAL operators. PRISM builds on this observation, using UDF outlining to generate LATERAL-free subqueries wherever possible, enabling SQL Server to unnest many more inlined UDFs (see Table 1) and achieve significantly improved performance.

[38] pioneered the first algorithm to unnest arbitrary subqueries (which DuckDB implements [40]), allowing the DBMS to unnest arbitrary inlined UDFs. Our experiments found that for some UDFs (i.e., Q17 from the ProcBench), unnesting the generated subquery results in orders of magnitude slower performance than naively evaluating the subquery row-by-row. Further research is necessary to unnest arbitrary subqueries without introducing regressions.

Optimizing Database-Backed Applications: A related research topic is lifting application logic into the DBMS. [3] use program synthesis to translate application code into SQL, reducing data movement between the application and the DBMS, and enabling the query optimizer to find better plans. [50] also explored program synthesis techniques to lift UDF code to SQL. Unfortunately, synthesis-based techniques lack termination guarantees, challenging their adoption in commercial systems [42].

In contrast, [10, 9] developed EqSQL, which uses static analysis techniques to translate application code into equivalent relational operations using a functional IR. Although EqSQL significantly improves performance, FROID chose a different translation strategy, which generates simpler code [42]. More recently, PyFROID [25, 36, 11] uses inlining-inspired techniques to translate queries written in pandas to SQL for faster query execution.

Cross-Language Optimization: Recent academic prototypes blur the lines between database systems and compilers. For instance, LingoDB [28, 27] uses MLIR [33] to represent relational and non-relational code as MLIR dialects, allowing cross-boundary optimizations. [17] adopt a similar approach, using GraalVM [49] to represent and optimize polyglot queries (i.e., queries invoking UDFs in multiple programming languages). Our techniques complement these designs, where the DBMS applies PRISM’s optimizations on its unified IR.

7. CONCLUSION

In this paper, we demonstrated how *UDF outlining* improves performance relative to conventional UDF inlining by selectively inlining only the portions of the UDF that are helpful for query optimization. By combining UDF outlining with four other complementary UDF-centric optimizations, our implementation (PRISM) achieves substantial speedups for UDF-centric queries running on DuckDB and SQL Server due to three benefits: more effective unnesting (sometimes resulting in over a 1000× speedup), improved data skipping (resulting in roughly a 10× speedup), and avoiding unnecessary joins (typically resulting a 1.02–1.62× speedup).

8. REFERENCES

- [1] V Aho Alfred, S Lam Monica, and D Ullman Jeffrey. *Compilers Principles, Techniques & Tools*. pearson Education, 2007.

- [2] F E Allen. Control flow analysis. *ACM Sigplan Notices*, 5(7):1–19, 1970.
- [3] A. Cheung, A. Solar-Lezama, and S. Madden. Optimizing database-backed applications with query synthesis. *ACM SIGPLAN Notices*, 48(6):3–14, 2013.
- [4] R. Cytron et al. An efficient method of computing static single assignment form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT POPL symposium*, pages 25–35, 1989.
- [5] E. Darling. When Does Scalar UDF Inlining Work In SQL Server? <https://erikdarlingdata.com/when-does-udf-inlining-kick-in/>, may 2022.
- [6] DuckDB. Group by never completes #9718. <https://github.com/duckdb/duckdb/issues/9718>, november 2023.
- [7] DuckDB. Overview of DuckDB Internals. <https://duckdb.org/docs/internals/overview.html>, april 2024.
- [8] M Elhemali et al. Execution strategies for sql subqueries. In *Proceedings of the 2007 ACM SIGMOD Conference*, pages 993–1004, 2007.
- [9] K Venkatesh Emani, Tejas Deshpande, Karthik Ramachandra, and S Sudarshan. Dbridge: Translating imperative code to sql. In *Proceedings of the 2017 ACM Conference*, pages 1663–1666, 2017.
- [10] K. Venkatesh Emani, Karthik Ramachandra, Subhro Bhattacharya, and S. Sudarshan. Extracting Equivalent SQL from Imperative Code in Database Applications. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1781–1796, San Francisco California USA, June 2016. ACM.
- [11] Venkatesh Emani, Avrilia Floratou, and Carlo Curino. Pyfroid: Scaling data analysis on a commodity workstation. 2024.
- [12] Steven Feuerstein. Speed up execution of your functions inside SQL statements with UDF pragma . <https://stevenfeuersteinonplsql.blogspot.com/2017/03/speed-up-execution-of-your-functions.html>, march 2017.
- [13] Y Foufoulas, A Simitsis, and Y Ioannidis. YeSQL: rich user-defined functions without the overhead. *Proc. VLDB Endow.*, 15(12):3730–3733, August 2022.
- [14] Yannis Foufoulas, Alkis Simitsis, Lefteris Stamatogiannakis, and Yannis Ioannidis. YeSQL: "you extend SQL" with rich and highly performant user-defined functions in relational databases. *Proc. VLDB Endow.*, 15(10), June 2022.
- [15] Kai Franz, Samuel I Arch, Denis Hirn, Torsten Grust, Todd Mowry, and Pavlo. Dear user-defined functions, inlining isn't working out so great for us. let's try batching to make our relationship work. sincerely, sql. In *CIDR 2024, Conference on Innovative Data Systems Research*, 2024.
- [16] César Galindo-Legaria and Milind Joshi. Orthogonal optimization of subqueries and aggregation. *ACM SIGMOD Record*, 30(2):571–581, 2001.
- [17] Philipp Marian Grulich, Steffen Zeuch, and Volker Markl. Babelfish: efficient execution of polyglot queries. *Proc. VLDB Endow.*, 15(2):196–210, October 2021.
- [18] S Gupta, S Purandare, and K Ramachandra. Aggify: Lifting the Curse of Cursor Loops using Custom Aggregates. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 559–573. ACM, June 2020.
- [19] S Gupta and K Ramachandra. Procedural extensions of SQL: understanding their usage in the wild. *Proc. VLDB Endow.*, 14(8):1378–1391, April 2021.
- [20] Ravindra Guravannavar and S Sudarshan. Rewriting procedures for batched bindings. *Proceedings of the VLDB Endowment*, 1(1):1107–1123, 2008.
- [21] Matthew S Hecht and Jeffrey D Ullman. Flow graph reducibility. In *Proceedings of the fourth annual ACM symposium on Theory of computing*, pages 238–250, 1972.
- [22] D Hirn and T Grust. One WITH RECURSIVE is Worth Many GOTOs. In *Proceedings of the 2021 International Conference on Management of Data*, pages 723–735, June 2021.
- [23] Denis Hirn. Decorrelation and parallelization of recursive and materialized CTEs #10357. <https://github.com/duckdb/duckdb/pull/10357>, feb 2023.
- [24] Denis Hirn. Apfel. <https://apfel-db.cs.uni-tuebingen.de/>, may 2024.
- [25] A Jindal et al. Magpie: Python at Speed and Scale using Cloud Backends. 2021.
- [26] Richard Johnson, David Pearson, and Keshav Pingali. The program structure tree: Computing control regions in linear time. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 171–185, 1994.
- [27] Michael Jungmair and Jana Giceva. Declarative sub-operators for universal data processing. *Proceedings of the VLDB Endowment*, 16(11):3461–3474, 2023.
- [28] Michael Jungmair, André Kohn, and Jana Giceva. Designing an open framework for query optimization and compilation. *Proceedings of the VLDB Endowment*, 15(11):2389–2401, 2022.
- [29] Won Kim. On optimizing an sql-like nested query. *ACM Transactions on Database Systems (TODS)*, 7(3):443–469, 1982.
- [30] Steffen Kläbe, Bobby DeSantis, Stefan Hagedorn, and Kai-Uwe Sattler. Accelerating python udfs in vectorized query execution. CIDR Conference, 2022.
- [31] T Kotzmann et al. Design of the java hotspot™ client compiler for java 6. *ACM TACO*, 5(1):1–32, 2008.
- [32] P-Å Larson et al. Sql server column store indexes. In *Proceedings of the 2011 ACM SIGMOD Conference*, pages 1177–1184, 2011.
- [33] C Lattner et al. Mlir: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM CGO*, pages 2–14, 2021.
- [34] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.
- [35] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. Morsel-driven parallelism: a numa-aware query evaluation framework for the

- many-core age. In *Proceedings of the 2014 ACM SIGMOD Conference*, pages 743–754, 2014.
- [36] X Liu, V Emani, A Floratou, J Cahoon, P Seamark, and C Curino. Polysem: Efficient polyglot analytics on semantic data. 2023.
 - [37] Microsoft. Scalar User-Defined Functions for In-Memory OLTP. <https://learn.microsoft.com/en-us/sql/relational-databases/in-memory-oltp/scalar-user-defined-functions-for-in-memory-oltp/>, june 2016.
 - [38] Thomas Neumann and Alfons Kemper. Unnesting arbitrary queries. *Datenbanksysteme für Business, Technologie und Web (BTW 2015)*, 2015.
 - [39] Oracle. Oracle9i Database New Features. https://docs.oracle.com/cd/A91202_01/901_doc/server.901/a90120/ch2_feat.htm, june 2001.
 - [40] Mark Raasveldt. Correlated Subqueries in SQL. <https://duckdb.org/2023/05/26/correlated-subqueries-in-sql.html>, may 2023.
 - [41] Mark Raasveldt and Hannes Mühleisen. Duckdb: an embeddable analytical database. In *Proceedings of the 2019 SIGMOD Conference*, pages 1981–1984, 2019.
 - [42] K Ramachandra, K Park, K. V Emani, A Halverson, C Galindo-Legaria, and C Cunningham. Froid: optimization of imperative programs in a relational database. *Proc. VLDB Endow.*, 11(4):432–444, December 2017.
 - [43] B K Rosen, M N Wegman, and F K Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT POPL*, 1988.
 - [44] P Seshadri, H Pirahesh, and TY C Leung. Complex query decorrelation. In *Proceedings of the Twelfth ICDE*, pages 450–458, 1996.
 - [45] V Simhadri, K Ramachandra, A Chaitanya, R Guravannavar, and S. Sudarshan. Decorrelation of user defined function invocations in queries. In *2014 IEEE 30th ICDE*, pages 532–543, Chicago, IL, USA, March 2014. IEEE.
 - [46] SingleStore. SingleStoreDB Cloud Release Notes. <https://docs.singlestore.com/cloud/release-notes/singlestoredb-cloud-release-notes/>, november 2021.
 - [47] L Spiegelberg et al. Tuplex: Data Science in Python at Native Code Speed. In *Proceedings of the 2021 SIGMOD Conference*, pages 1718–1731. ACM, June 2021.
 - [48] L F. Spiegelberg and T Kraska. Tuplex: robust, efficient analytics when Python rules. *Proc. VLDB Endow.*, 12(12):1958–1961, August 2019.
 - [49] T Würthinger et al. One vm to rule them all. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*, pages 187–204, 2013.
 - [50] G Zhang, Y Xu, X Shen, and I Dillig. UDF to SQL translation through compositional lazy inductive synthesis. *Proc. ACM Program. Lang.*, 5(OOPSLA):1–26, October 2021.

Technical Perspective on “MEMPHIS: Holistic Lineage-based Reuse and Memory Management for Multi-backend ML Systems”

Arun Kumar

University of California, San Diego and RapidFire AI, Inc.
akk018@ucsd.edu

ML deployments in real-world settings are often heterogeneous, spanning local multi-core CPU operations, GPU-accelerated computations, and distributed execution on platforms such as Apache Spark. This heterogeneity arises from practical necessity: a feature engineering step might run locally, a large matrix multiplication offloads to Spark when data exceeds driver memory, and DNN layers execute on GPUs. Or an initial homogeneous setup might evolve over time into a heterogeneous one. This multi-backend reality creates a systems challenge that has received surprisingly little research attention: *How to holistically manage computation reuse and memory across backends with fundamentally different execution models, memory hierarchies, and caching primitives?*

The Problem and Prior Work. Exploratory data science workloads often involve redundant computations; hyperparameter tuning, cross-validation, and AutoML repeatedly execute similar computations with slight variations. Prior work at the data+ML systems intersection proposed reuse techniques but largely in isolation: coarse-grained pipeline-level reuse via compile-time materialization (e.g., HELIX), input data pipeline caching for DNNs (e.g., tf.Data, Cachew), prediction caching for inference (Clipper), and DNN feature layer reuse (e.g., Vista, Nautilus). These target specific workload-backend combinations and do not compose well. The authors’ prior LIMA framework introduced fine-grained lineage-based reuse but was limited to local CPU operations. In production deployments supporting multiple teams, be it at Big Tech, enterprises, or academic research centers, I have seen how ML/AI users often struggle with such fragmentation, manually orchestrating caching across backends with ad-hoc scripts.

The MEMPHIS Approach. This paper by Phani and Boehm, which received the EDBT 2025 Best Paper Award, introduces a new systems approach they call MEMPHIS. The key insight is that fine-grained lineage tracing, i.e., tracking provenance of every intermediate at the operator level, can serve as a unified abstraction that transcends backend boundaries. MEMPHIS maintains a hierarchical lineage cache mapping traces to backend-specific objects: in-memory matrix blocks on the driver, distributed RDDs in Spark, and GPU pointers on accelerators.

The technical contributions address each backend’s distinct challenges. For Spark, lazy evaluation means eager caching degrades performance; the paper shows eager materialization of 12K RDDs is 10× slower than no caching. MEMPHIS introduces delayed caching that defers persistence until repeated hits, lazy garbage collection for dangling broadcast references, and driver-side action reuse to bypass redundant job execution. For GPUs, allocation and copy overhead dominate (4.6× and 9× longer than computation respectively). MEMPHIS combines reuse with memory recycling using

Live/Free pointer lists, prioritizing exact-size recycling to avoid fragmentation while using a scoring function based on recency, lineage height, and compute cost. The compiler is extended with asynchronous prefetch operators, workload-aware eviction injection, and an operator linearization algorithm to maximize concurrent cross-backend execution.

Practical Significance. What makes this work compelling is its comprehensive systems approach and empirical rigor. The authors tackle full integration challenges: how lineage tracing interacts with Spark’s BlockManager, how GPU eviction must account for CUDA synchronization barriers, and how compiler rewrites inject checkpoints based on estimated reuse potential. This requires not just deep understanding of each system’s tradeoffs but also synthesizing across them to make the approach work.

Implemented in Apache SystemDS and evaluated on 7 diverse ML tasks—grid search (9.6× speedup), Poisson matrix factorization (7.9×), Hyperband, data cleaning pipelines, dropout tuning, machine translation, and transfer learning—comparisons against PyTorch, LIMA, HELIX, and others show consistent improvements. The open-source implementation and reproducibility package strengthen practical value.

Limitations and Future Directions. MEMPHIS is deeply integrated with SystemDS’s compilation model. Extending to other setups, e.g., Ray-based, would require heavy-duty engineering. The paper also focused on multi-core CPU and Spark backends for clusters, which is a great start, but multi-GPU clusters are also common, as are newer accelerators such as TPUs—all this merits further study. Cross-backend object migration was implemented but data movement overhead outweighed benefits. At the algorithmics level, operator scheduling remains heuristic-based; cost-based reuse-aware placement is future work. Overall, there are many opportunities for future research across the systems stack building on this foundation.

Broader Context. MEMPHIS demonstrates that holistic reasoning about reuse across the full software stack yields substantial benefits over optimizing components in isolation. This work reflects a major trend: as ML/AI infrastructure grows in complexity and scale, core database systems principles—query optimization, tiered caching, memory management, lineage tracking, etc.—become critical for efficient operation at scale. For instance, it is not a coincidence that OpenAI’s first acquisition was a DBMS startup. I hope the DB community keeps pursuing such cross-pollination to power the ongoing AI boom while mitigating resource/energy wastage.

Lineage-based Reuse and Memory Management for Multi-backend ML Systems

Arnab Phani
TU Berlin & BIFOLD
Germany
arnab.phani@tu-berlin.de

Matthias Boehm
TU Berlin & BIFOLD
Germany
matthias.boehm@tu-berlin.de

ABSTRACT

Modern machine learning (ML) systems leverage multiple backends, including CPUs, GPUs, and distributed execution platforms like Apache Spark or Ray. Depending on workload and cluster characteristics, these systems typically compile an ML pipeline into hybrid plans of in-memory CPU, GPU, and distributed operations. Prior work found that exploratory data science processes exhibit a high degree of redundancy, and accordingly applied tailor-made techniques for reusing intermediates in specific backend scenarios. However, achieving efficient holistic reuse in multi-backend data systems remains a challenge due to its tight coupling with other aspects such as memory management, data exchange, and operator scheduling. In this paper, we introduce MEMPHIS, a principled framework for holistic, application-agnostic, multi-backend reuse and memory management. MEMPHIS's core component is a hierarchical lineage-based reuse cache, which acts as a unified abstraction and manages the reuse, recycling, exchange, and cache eviction across different backends. To address challenges of different backends such as lazy evaluation, asynchronous execution, memory allocation overheads, small available memory, and different interconnect bandwidths, we devise a suite of cache management policies. Moreover, we extend an optimizing ML system compiler by special operators and rewrites for asynchronous data exchange, workload-aware speculative cache management, and related operator ordering for concurrent execution. Our experiments across diverse ML tasks and pipelines show improvements of up to 9.6x compared to state-of-the-art ML systems.

1. INTRODUCTION

Modern ML systems are widely used for model training, inference, as well as data preparation and feature transformations of multi-modal data [70]. Data scientists hierarchically compose complex ML pipelines from black-box primitives [15]. The exploratory nature of these pipelines causes high computational redundancy [71, 87, 23, 39].

Sources of Redundancy: This high computational redundancy arises from various sources including incremental modifications of ML pipelines and hyper-parameter tuning,

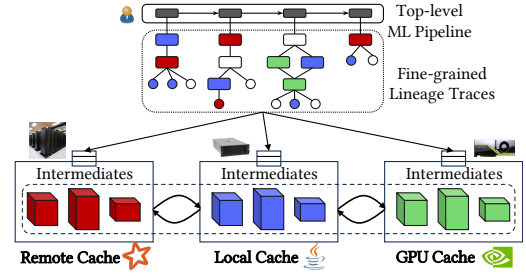


Figure 1: Multi-backend Reuse Cache for Intermediates.

as well as fine-grained redundancy in training and inference[71]. ML libraries construct high-level primitives from smaller built-ins (e.g., scikit-learn's `make_pipeline`), hiding the independent operations, causing unnecessary redundancy. AutoML systems [46, 22, 28, 49, 78] and ML agents [61, 27] combine tasks like data cleaning, feature engineering, hyper-parameter tuning, and model training [45, 44], and explore pipelines with slight changes. Deep neural network (DNN) workloads repeatedly execute data pipelines [59, 55, 33] and forward paths [60, 73] at batch granularity, while inference frameworks for object detection and machine translation [17] encounter duplicate inputs [19, 43, 86].

Multi-backend ML Systems: ML pipelines with diverse data and workload characteristics necessitate multiple, specialized backends for achieving efficiency and scalability. Example pipelines include (1) hybrid local/distributed runtime plans for large-scale ML [89, 14, 51, 74], (2) large-scale data validation [76, 72] and cleaning [78, 79], (3) sampling, feature selection [15], and data augmentation [20, 7], which iteratively change data sizes by orders of magnitude, (4) exploratory ML algorithm research [52], (5) model training on structured and unstructured features [60], (6) AutoML systems [14, 15, 46] supporting diverse ML algorithms, and (7) model debugging [48, 75] with configuration-dependent intermediate sizes. These tasks are often combined into complex pipelines, increasingly fostering the development of ML systems with optimizing compilers, specialized operator placement strategies [15, 46, 12], and multiple backends including local CPU/GPU/FPGAs, distributed MapReduce/Spark/Dask/Ray and federated backends [41, 10]. Example systems include PyTorch [64] and TensorFlow [1] (which leverage CPU, GPU), MLlib [89] and Dask-ML [74] (which utilize local and distributed), and SageMaker [46] (which utilizes CPU, GPU, and Spark). Additionally, unified data analytics frameworks [12, 24, 31], polyglot [3, 30, 29], federated [41, 10], and composable [56] data management systems support cross-platform runtime backends.

© Copyright held by the owner/author(s). This is a minor revision of the paper "MEMPHIS: Holistic Lineage-based Reuse and Memory Management for Multi-backend ML Systems" published in EDBT 2025, March 25-28 on OpenProceedings.org. DOI: <https://doi.org/10.48786/edbt.2025.21>

Challenges in Multi-backend Reuse: Introducing a static reuse cache for redundancy elimination into multi-backend systems—as shown in Figure 1—poses major challenges due to heterogeneous backend characteristics. These backends differ in execution models (eager, lazy, asynchronous), memory characteristics (large distributed, small on-chip), data exchange bandwidths, other backend-specific properties like GPU memory allocation overhead, and target workloads, ranging from pre-processing to mini-batch DNNs and LLMs. To address this diversity, modern data systems employ various techniques for memory management [67, 88, 4, 37], operator placement [53, 8, 9], data exchange [59, 33, 91, 83, 38], and parallelization [92, 26, 62, 77] tailored to specific workloads and backends. This heterogeneity necessitates a principled approach for efficient reuse and cache integration across diverse compilation, memory management, and operator scheduling techniques, which is currently lacking.

Existing Work on Reuse: There exists extensive research on reusing query intermediates in database systems [36, 93]. In ML, prior work on reuse rely on coarse-grained reuse of the top-level primitives (see Figure 1) [87, 23, 85, 90, 81, 39]. However, this black-box view of individual ML stages fails to handle the ubiquitous fine-grained redundancy (e.g., repeated matrix multiplications inside/across primitives) and non-determinism (randomized primitives). The LIMA framework [71] introduced fine-grained reuse, leveraging lineage traces on individual operations and functions to uniquely identify reusable intermediates, but was limited to local CPU operations. Prior work on application-specific, multi-backend reuse includes heuristics-based Spark RDD caching [14, 6, 33], input data pipeline reuse [59, 33, 55], prediction caching [19, 43], and GPU-CPU activation offloading for DNNs [66, 35, 50]. We refer the interested reader to a broader discussion of related work [69]. These approaches are tailored to workload-backend combinations, and fail to eliminate redundancy in modern data-centric ML pipelines.

MEMPHIS Overview and Contributions: We introduce MEMPHIS, a *holistic framework* for multi-backend reuse of intermediates and memory management. Key principles are (1) unified cache abstraction with multi-backend data objects, (2) backend-specific cache and memory management, and (3) robust integration with the compiler and runtime to support diverse workloads. MEMPHIS is fully integrated into Apache SystemDS¹ [15], and utilizes *three* representative backends: in-memory operations (SystemDS), distributed operations (Spark), and GPUs, while being applicable to diverse system architectures. MEMPHIS extends LIMA’s [71] lineage-based reuse framework with novel compiler and runtime techniques for reusing Spark and GPU intermediates, handling Spark’s lazy evaluation and small GPU memory. The introduced techniques also generalize to reusing intermediates in column-at-a-time or vector-at-a-time query processing with multiple backends. In the following, we discuss the relevant background, introduce our multi-backend lineage tracing framework with runtime cache management and compiler optimizations, and discuss the generalizability of these techniques across workloads and systems.

2. BACKGROUND

This section describes the necessary background of ML system internals, Spark and GPU backends, their execu-

tion models and memory management. The CPU, GPU, and Spark backends differ substantially in their execution model, memory management, and target workloads.

Program/DAG Compilation: We categorize the optimization scope of ML systems into three types [16]. *Eager execution* libraries like NumPy [84] and Scikit-learn [65] execute operations immediately and rely on Python for control flow. *DAG compilation* systems like TensorFlow [1, 11] and PyTorch [64] perform lazy evaluation over a compute DAG, enabling a larger optimization scope. Finally, *program compilation* in Julia [13] and SystemDS [15] (previously SystemML [14]) compile scripts into program block hierarchies, where last-level blocks comprise operator DAGs. Compiler rewrites like common subexpression elimination (CSE) and code motion fail to eliminate all redundancy, as conditional control flow is often unpredictable at compile time.

Operator Scheduling: An operator scheduler converts operator DAGs into backend-specific instruction streams and has two key responsibilities: (1) *operator placement*, which minimizes execution time in multi-backend runtimes using heuristics or configurations, and (2) *operator linearization*, which orders operator DAGs into sequential or parallel instruction streams, affecting parallelism and memory requirements. SystemDS linearizes the DAGs depth-first and places operations with higher memory estimates to Spark and compute-intensive, dense operations to GPU. Figure 2(a) shows the lifecycle of a data object across backends.

Spark’s Execution and Memory Model: Spark follows a driver-executor architecture: the driver manages scheduling and local execution, while executors run distributed tasks. Computation is expressed using RDDs (Resilient Distributed Datasets) [89], representing partitioned collections of matrix/frame tiles manipulated through lazily evaluated *transformations*. An *action* invokes the **DAGScheduler** to construct a job DAG of stages separated by shuffle boundaries. Spark’s unified memory model divides heap space into execution and storage (default 60%), allowing dynamic sharing between computation, temporary data, caching, and broadcasting. Shuffle outputs and broadcast data are implicitly cached, while explicit `persist()` enables reuse across jobs at different storage levels. Under memory pressure, cached partitions are evicted and recomputed from lineage.

Lazy Evaluation and Broadcast Challenges: Lazy evaluation presents several challenges for efficient reuse. First, eager caching (LIMA, `tf.Data` [59], `Cachew`) materializes RDDs after each instruction, triggering repeated jobs. In Figure 2(c), materializing 12K RDDs (4K reusable) is 10x slower than no caching. Second, caching all RDDs at runtime inflates cluster memory (20x in Figure 2(c)), requiring speculative caching. Finally, although broadcast operators are cheaper than shuffle joins, Spark’s lazy execution delays data transfer: **TorrentBroadcast** splits serialized data into 4 MB chunks stored in the driver and lazily distributed to executors. Consequently, broadcast data and dependent RDDs must be retained until job completion, creating *dangling references* that consume memory. Figure 2(b) shows linearized instructions (SP denotes Spark, CP denotes CPU) and the lineage graph for $b = (\mathbf{y}^\top \mathbf{X})^\top$, broadcasting $\mathbf{y}^\top$. The second transpose collects the vector $\mathbf{b}$ to the driver. The serialized $\mathbf{y}^\top$ remains in the driver’s memory until the job finishes, which is dependent on other operators’ linearization order.

GPU Execution Model and Memory Challenges: GPUs offer high peak performance and memory bandwidth, suit-

¹SystemDS: <https://github.com/apache/systemds>

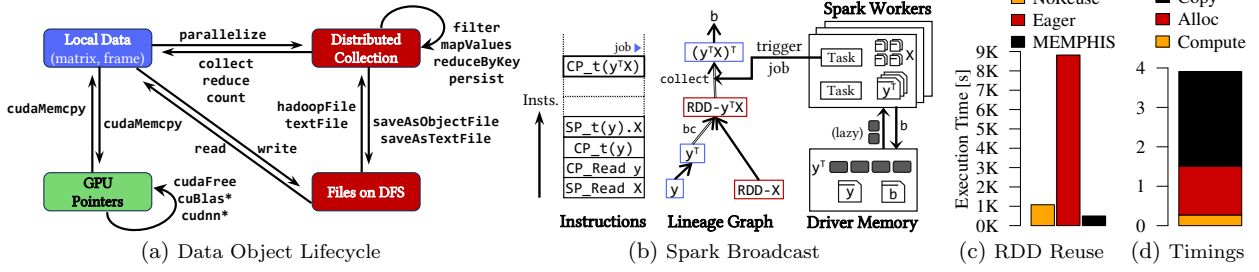


Figure 2: Spark and GPU Backend Details.

able for regular data access (e.g., DNNs). In SystemDS, CUDA kernels execute eagerly and sequentially within a stream but asynchronously with respect to the host, enabling CPU-GPU overlap. However, device-host transfers and memory deallocation introduce synchronization barriers that stall the CPU until pending kernels complete. Traditional eager caching also forces synchronization and degrades performance. As shown in Figure 2(d), for an affine layer with ReLU over 10 epochs of 1K mini-batches (128 rows), memory allocation/free and data transfer incur 4.6x and 9x higher overhead than computation, highlighting that static cache memory and traditional eviction policies, which increase memory pressure and data copy overhead, are unsuitable for GPU pointer caching.

3. HIERARCHICAL LINEAGE CACHE

The basic architecture of MEMPHIS with lineage tracing and a hierarchical cache is shown in Figure 5. The driver/host process handles the lineage tracing, compiler optimizations, and eviction planning, whereas the actual cached objects reside in the respective backends. MEMPHIS builds upon LIMA’s [71] basic lineage tracing (eager caching), and extends it with a hierarchical lineage cache and unified API for reuse across local, Spark, and GPU. This section describes our fine-grained lineage tracing, and the unified intermediate cache for backend-specific objects.

3.1 Supported API

MEMPHIS provides a set of system-internal methods to enable lineage-based reuse and simplify integration with ML systems across different backends, compilation, and operator placement. Specifically, `TRACE(inst)` captures operator lineage; `SERIALIZE(trace)` and `DESERIALIZE(log)` convert between in-memory lineage traces and lineage logs; `RECOMPUTE(log)` reproduces results from a log; `REUSE(trace)` skips execution by reusing cached results when available; `PUT(trace, object)` inserts instruction outputs into the backend-specific cache; and `MAKE_SPACE(object)` evicts cached objects to free backend memory.

3.2 Backend-agnostic Lineage Tracing

A lineage trace—incrementally built at runtime—is a DAG with nodes and edges representing operations and data dependencies. We invoke `TRACE` before executing each linear algebra instruction (see Figure 3) and maintain a hash map (`LineageMap`) that maps live variable names to lineage DAGs. Each lineage item captures the opcode, data items, and pointers to input lineage items, and every `TRACE` call creates a new item that is added to the map. To enable effi-

cient reuse, lineage items implement `hashCode` and `equals`, where hashes are computed from input item hashes, opcodes, and data items, and equality checks use a non-recursive, queue-based traversal with sub-DAG memoization and early aborts. We further support serializing lineage traces and exact recomputation via `RECOMPUTE`, which re-applies the full compilation chain to regenerate identical results. Beyond reuse, fine-grained lineage tracing and these APIs provide a general infrastructure for debugging, interpretability, recomputation, model versioning, and lifecycle management, and integrate naturally with diverse data systems.

3.3 Multi-backend Lineage Cache

We leverage that our lineage uniquely identifies intermediates to enable reuse of previously computed results. The lineage cache serves as a repository for these lineage traces, maintaining the necessary data structures to efficiently map them to their corresponding backend-specific data objects.

Lineage Cache: The lineage cache is a hash map that maps lineage items to cached data objects (see Figure 5 middle). Figure 3 shows the pseudo-code of the reuse logic integrated in the main instruction execution path, which seamlessly applies to all instructions. We call `REUSE` for each instruction. If the output exists in the cache, we reuse the data object, assign it to the live variable, and skip the instruction. Otherwise, we execute the instruction and store the output in the cache via `PUT`. Additionally, the lineage cache entries hold metadata including compute costs, access counts, and status.

```
while (inst in insts)
  lt = TRACE (inst)
  if (!REUSE (lt))
    out = exec(inst)
    PUT (lt, out)
```

Figure 3: Reuse API.

Redundancy in Lineage Items: Generating a new lineage item for each instruction *before* reuse creates redundant lineage DAGs across `LineageMap` and lineage cache entries. To address this issue, upon successful probes, we replace the respective `LineageMap` entries with the lineage keys of cached objects. As Figure 4 shows, this compaction increases shared sub-DAGs (objects with identical references), which in turn improves probing efficiency and reduces the memory footprint.

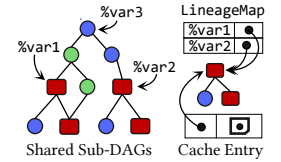


Figure 4: Compaction.

Backend-local Cached Objects: The lineage cache entries are wrappers around backend-specific pointers. These pointers refer to in-memory matrix blocks, scalars, distributed RDDs, GPU objects, and disk-evicted binaries. This design allows seamless data movement like broadcasting local inputs to remote backends, transferring Spark job results to

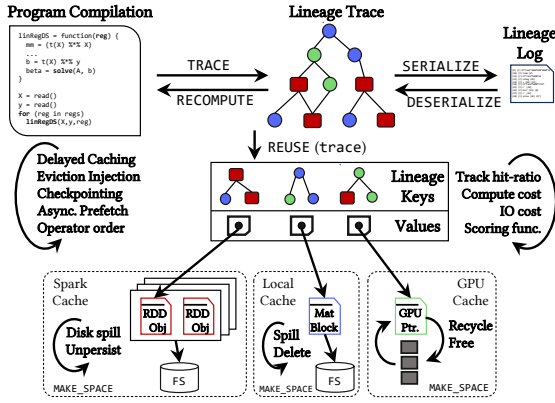


Figure 5: Hierarchical Lineage Cache & Reuse Overview.

the driver, and GPU-to-host copies. Moreover, the wrappers enable caching the same object in multiple backends (beneficial for data-local scheduling) and seamless extension to new backends. The centralized lineage tracing and reuse facilitates a reuse-aware compilation to increase reuse potential and reduce caching overheads. Figure 5 summarizes the lineage tracing lifecycle, reuse operations, backend-local evictions, and compiler extensions.

Multi-level Reuse: MEMPHIS reuses outputs of deterministic functions and basic blocks (code block w/o control flow) when called repeatedly with the same inputs, even when the inputs and outputs are scattered across different backends. We use a special lineage item containing the function name and inputs for each function output [71]. These items are managed alongside regular entries and are subject to eviction by the respective backends. Combining coarse-grained reuse (e.g., function reuse) with fine-grained reuse (e.g., operator-at-a-time reuse, as shown in Figure 3) is advantageous for hierarchically-composed data-centric ML pipelines. Multi-level reuse effectively reduces fine-grained remote operations, related data exchange, and cache pollution (avoids caching large distributed objects, or many small GPU objects) and hence is resilient against cache evictions.

4. RUNTIME MULTI-BACKEND REUSE

Backend-local caches require tailored cache management due to their diversity. This section describes reuse and memory management for Spark and GPU (more details [69]).

4.1 Reuse & Memory Management in Spark

We use Spark’s caching API to cache RDDs in cluster memory. Furthermore, we introduce memory management and cache eviction tailored for Spark’s lazy evaluation.

Reuse Spark Actions: Spark actions trigger job execution and return the results to the driver. ML systems implicitly leverage these actions within distributed physical operators (e.g., single-block aggregates calling `reduce()` instead of `reduceByKey()`). Additionally, operator placement decisions may explicitly move data to the driver using `collect()`. Before triggering a Spark job, we reuse the previously collected result if available in the driver’s cache, and bypass the job execution. In Figure 6 (top entry), the `transpose` operator collects the vector `b` to the driver and stores it in the cache for future reuse. Reusing redundant actions in the driver

eliminates distributed operations, unnecessary distributed caching, and data exchange such as shuffle and collect.

Reuse RDDs: RDDs are distributed data collections, which are cached and reused in the cluster. Before executing a Spark transformation, the REUSE API probes the lineage cache and reuses the RDD (serves as a pointer to the distributed collection) if a match is found. Otherwise, we mark the RDD for caching with `persist()` and store it in the cache along with related metadata. `persist()` is a lazy operation—and thus, the cached RDD may not be materialized in Spark memory by the time of reuse. However, we reuse even unmaterialized RDDs to enhance compute-sharing across jobs, and enable shuffle-file reuse (data exchange optimization). Figure 6 (bottom entry) shows a cached RDD entry for $X^T X$, where X is distributed.

Lazy Garbage Collection: To tackle memory overhead caused by dangling RDDs and broadcast references (Figure 2(b)), we track internal metadata for each RDD, including #child RDDs, #pending consumers, materialization status, and its memory overhead. During reuse, `MAKE_SPACE` recursively traverses child RDDs and broadcasts, updating the metadata and clearing stale RDDs (not in use and already materialized). For example, in Figure 6 (bottom), we delete X^T and X once $X^T X$ is materialized. We use Spark’s `destroy` and `getRDDStorageInfo` for broadcast variable deletion and materialization checks. Furthermore, to mitigate caching overhead, the MEMPHIS compiler often defers RDD caching (*delayed caching* discussed in 5.2). Delayed caching along with Spark action reuse further increases dangling references. In Figure 6 (top), $y^T X$ RDD caching is delayed and remains unmaterialized due to the reuse of the collected `b` at the driver. Such RDDs prevent the garbage collection of their child RDDs (i.e., y^T , X). After k cache misses, we asynchronously trigger a Spark job (calling `count()`) to materialize such an RDD. Any subsequent reuse then cleans up the child RDDs. These lazy cleanups ensure periodic reclamation of driver and cluster memory without hindering reuse.

EXAMPLE 1 (GRID SEARCH LINEAR REGRESSION).

We apply grid search hyper-parameter tuning on a direct-solve linear regression (`linRegDS`) with distributed features X and local responses y . Figure 7 shows the `linRegDS` function, which is invoked for a list of regularization parameters (`reg`). The core operations $X^T X$ and $X^T y$ are independent of `reg` and thus, reusable across calls. `SystemDS` compiles $X^T X$ as a shuffle-based matrix multiplication and rewrites $X^T y$ to $(y^T X)^T$ for broadcast-based multiplication. The second transpose of $(y^T X)^T$ and the `solve` trigger Spark jobs. Figure 7 shows the DAG for `linRegDS`. Rectangles represent distributed operations. Reusable distributed and local operations are colored red and blue, and data exchange is indicated by double lines. The first `linRegDS` call creates lineage cache entries for all operations and caches in-memory matrix outputs of local operations. The second transpose of $(y^T X)^T$ caches the collected column vector `b` as an in-memory matrix in the driver. Our compiler enables delayed caching for RDDs, e.g., deferring caching until the second cache hit. Accordingly, the second `linRegDS` call marks the RDDs (`mm`, $y^T X$) for distributed caching, reuses the local operations and the collected `b`. Reusing the

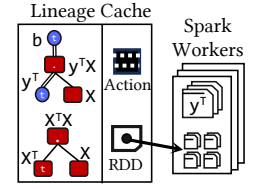


Figure 6: Spark Reuse.

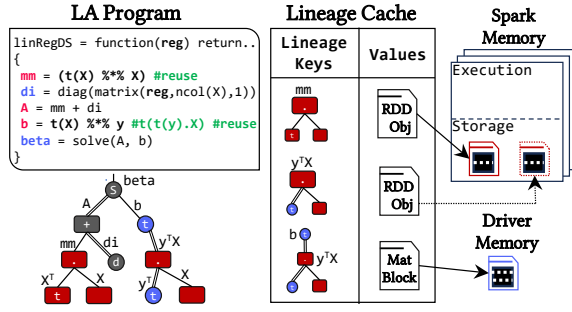


Figure 7: Spark Action and RDD Reuse Example.

Spark action result **b** entirely eliminates the need to trigger the Spark job. However, not triggering the job keeps the $y^T X$ RDD unmaterialized in the executors. The third call reuses the **mm** RDD, and subsequent calls clean up its child RDDs. After k (default three) cache misses for $y^T X$ RDD, we asynchronously materialize the $y^T X$ RDD (dotted line).

4.2 Reuse & Memory Management in GPU

To manage limited GPU memory and allocation overhead, we combine reuse and recycling in a unified memory manager with moving boundaries that reuses **pointers**. Figure 8 depicts the memory management operations.

Live Variable Management: All pointers from allocation to deallocation are managed by the lineage cache. Figure 8(a) shows the lifecycle of a GPU pointer. We organize the allocated pointers in two lists: a **Live** and a **Free** list. The **Live** pointers correspond to variables which are still in use (i.e., pending consumers). The **Free** list comprises a hash map that maps sizes to a priority queues of free pointers of the respective sizes. Before executing an operation, we allocate memory for the output via `cudaMalloc` and place the allocated pointer in the **Live** list. After the last use, the pointers are moved to the **Free** list, as shown in Figure 8(b). This strategy of maintaining free memory pools benefits mini-batch processing with fixed batch sizes.

Reuse: Lineage cache entries encapsulate GPU pointers and metadata like data characteristics. Before executing an instruction (composed of one or more GPU kernels), the **REUSE** call reuses a cached output pointer, if available, and skips kernel launching. As Figure 8(c) shows, the reused pointer is moved from **Free** to **Live** list. Reusing a pointer may cause multiple variables referencing a single pointer. We track the *reference count* for each pointer indicating the #variables referencing it, and only when the reference count becomes zero, the pointer is returned to the **Free** list.

Memory Recycling: Pointers in the **Free** queues are ordered by a scoring function. Once the GPU memory is full, we start recycling the free pointers as a form of eviction (Figure 8(d)). We first look for a free pointer with the exact size to recycle. If unavailable, we free a pointer just larger than the required size. If all free pointers are smaller than the required size, we repeatedly free a pointer until `cudaMalloc` is successful. If allocation still fails, we clear all free pointers. Even then, the allocation may fail due to many live variables and memory fragmentation. In such cases, we initiate the device-to-host eviction process, and finally a full defragmentation, but this event is rare in practice. Memory recycling benefits typical DNN workloads with repeated operations on

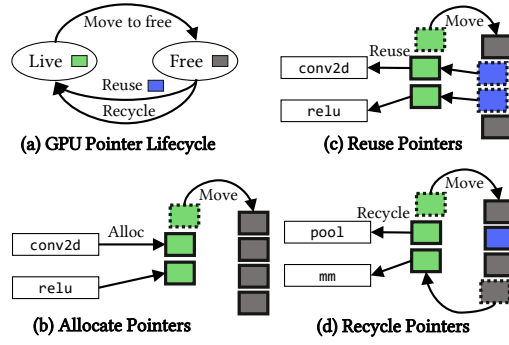


Figure 8: Reusing and Recycling of GPU Pointers.

fixed-sized intermediates, such as the forward and backward passes of mini-batch processing. The primary objective of these steps is to avoid memory allocation, deallocation, and fragmentation, without compromising the reuse potential.

4.3 Cache Eviction

We employ backend-specific, cost-based eviction policies to remove intermediates with low reuse potential. For Spark, we extend the Cost&Size policy [23, 71] to account for lazy evaluation by evicting cached RDDs, which is orthogonal to Spark’s partition-level eviction. We heuristically allocate 80% (configurable) of Spark’s storage memory for reuse, reserving the remainder for broadcast variables and compiler-placed checkpoints (Section 5.2). Each cached RDD o is ranked using analytical compute cost $c(o)$, estimated worst-case size $s(o)$, and reuse statistics ($\#hits\ r_h$, $\#misses\ r_m$, $\#jobs\ r_j$), with the eviction scoring function

$$\arg \min_{o \in Q} (r_h(o) + r_m(o) + r_j(o)) \cdot c(o)/s(o), \quad (1)$$

where Q is a priority queue of cached RDDs. The **MAKE_SPACE** method marks RDDs for asynchronous eviction and refreshes cache metadata using `getRDDStorageInfo`, with temporary overflows efficiently handled via Spark’s partition spilling. For GPUs, the eviction policy determines the order in which pointers are recycled or freed from free queues and is tailored to mini-batch workloads. The eviction scoring function for a cached GPU object o is

$$\arg \min_{o \in Q} T_a(o) + 1/h(o) + c(o) \quad (2)$$

where $T_a(o)$ denotes the normalized last access time to preserve recently reused pointers, $h(o)$ is the lineage DAG height favoring reuse of *input data pipelines* [55], and $c(o)$ is the estimated compute cost, prioritizing recycling of inexpensive intermediates (e.g., ReLU before Conv2d).

5. COMPILER INTEGRATION

Runtime-level reuse and memory management are valuable, but holistic *compiler- and runtime-level* handling yields additional benefits. In this section, we describe optimizations for operator parallelism, data exchange, workload-aware cache management and checkpoint placement, and an operator linearization strategy to increase parallel execution.

5.1 Asynchronous Remote Jobs

To enable inter-backend parallelism and asynchronous data transfer, we introduce new operators and rewrites.

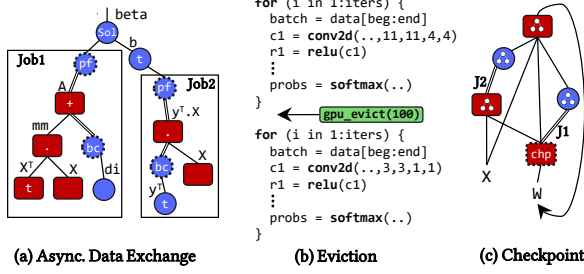


Figure 9: Compiler-assisted Reuse.

First, we add a **prefetch** operator that triggers remote (Spark/GPU) jobs and asynchronously fetches results without blocking the CPU instruction stream. The corresponding rewrite identifies operators that trigger remote jobs via **collect** or **cudaMemcpyDeviceToHost**, treats them as roots of remote operator chains, and inserts **prefetch** instructions after these roots. Additionally, this rewrite flags all other Spark actions for asynchronous execution. At runtime, the operator scheduler launches these operations asynchronously and returns **future** objects [58], enabling concurrent execution of Spark jobs, GPU kernels, and local operators. Figure 9(a) shows the DAG from Example 1 after **prefetch** placement (*pf*) for two jobs. Since immediate caching is infeasible for asynchronous execution, the main thread propagates lineage traces to prefetch threads, allowing fetched results to be cached (**PUT**) once available and reused across iterations. The **prefetch** instruction of Job2 in Figure 9(a) caches the fetched result of $y^T X$. Second, we introduce an asynchronous **broadcast** operator to optimize local-to-remote data transfer, which the compiler places at the end of local operator chains (see *bc* operators in Figure 9(a)).

5.2 Workload-aware Cache Management

We introduce new cache management operators and placement rewrites to reduce caching overhead and improve the utility of reuse and memory management.

Eviction Injection: Our eviction logic, (in Section 4.3), operates incrementally, evicting one object-at-a-time to make space for new objects. This approach incurs high eviction overhead during allocation pattern shifts, common in GPU workloads. Figure 9(b), shows an ensemble learning script utilizing two models (AlexNet & VGG16) for joint prediction. In the first **for** loop, MEMPHIS efficiently recycles the less reusable pre-allocated pointers. However, the allocation pattern shifts in the second loop due to varying **conv2d** kernel sizes, leading to high allocation-deallocation overhead and memory fragmentation. To mitigate this issue, we introduce an **evict** instruction for cache cleanup. A program-level rewrite identifies these loop patterns and injects **evict** instructions with arguments of cache size to clean up. At runtime, the **evict** instruction utilizes the backend-specific eviction logic to free up space. The rewrite avoids full eviction if access patterns repeat. Other DNN frameworks require manual placement of such cleanup [18].

RDD Checkpoint Placement: Lineage-based reuse eliminates redundancy across Spark jobs, but Spark’s lazy evaluation introduces another form of redundancy due to shared dataflow dependencies between jobs. Prior work [6, 14] proposes, rule-based checkpoint (**persist**) placements. Here, we introduce two workload-aware rewrites for checkpoint placement. (1) The first rewrite identifies overlapping Spark jobs

within a basic block and injects a checkpoint after the last shared operator. (2) The second rewrite targets common iterative algorithms, where updated variables create increasingly large operator graphs. Figure 9(c) shows a simplified DAG of one factor matrix **W** of Poisson Non-negative Matrix Factorization [47]. Here, each node represents a Spark (red) or local (blue) operator chain. Every iteration updates **W** and triggers two jobs (J1, J2), both lazily executing all previous iterations. Our rewrite identifies the variables that are updated in each iteration and places checkpoints to reuse previous iterations’ results. In this example, we cache **W** (indicated by *chp*) in each iteration. For a seamless integration, at runtime, the lineage cache eviction tracks these checkpointed RDDs and updates the cache metadata.

Delayed Caching: Caching incurs a probing cost and other backend-specific overheads, especially for long-running, complex workloads. Caching non-repeating large RDDs increases memory pressure and garbage collection overhead on both executors and drivers. Similarly, DNN training with no reuse potential in GPUs suffer from allocation overhead, memory fill-up, and fragmentation. Based on the observation that reusable operations repeat multiple times for hierarchically composed ML pipelines, we introduce *delayed caching*, that defers caching until the *n*-th (*delay factor*) cache hit. At runtime, the **PUT** call creates an empty lineage cache entry with status **T0-BE-CACHED** upon the first cache hit. If the operator repeats *n* times, we put the actual object in the cache and change the status to **CACHED**. Together with our cache eviction schemes, cache entry strategies such as delayed caching substantially reduce the overhead and cache misses.

Automatic Parameter Tuning:

We introduce a program-level rewrite for tuning the *delay factor* *n* (no delay is *n* = 1) and the *Spark storage level* (for RDD caching) of each basic block, based on estimated reuse potential. This rewrite recursively traverses all program blocks, analyzing the execution frequency (nested loops, function calls) and the presence of loop-dependent operations (not reusable). A second pass then assigns the values for *n* (*n* = 1 if >80% reusable) and storage level, and stores them in the block headers.

```
# Feature selection
for (i in 1:ncol(X)) {
  Xi = cbind(X_g, X[,i])
  ac = lm(Xi,...)
  ..check if best AIC..
  X_b = cbind(X_b, X[,i])
}
# Hyperparameter tuning
for (lambda in lambdas)
  model = TrainPipe(lambda)
# Clean and train
TrainPipe = function(lambda) {
  X_cl1 = imputeMV(X_b)
  X_cl2 = outlierIQR(X_cl1)
  while(minimize)
    ..training loop..
}
```

Figure 10: Delay Factors.

Figure 10 shows a *simplified* ML pipeline where, operations in block 1 (step-wise feature selection [2]) are loop-iteration-dependent (X_i varies with *i*) and thus, are deemed not reusable (*n* = 4). In reusable (*n* = 1), cleaning and outlier removal methods **imputeMV** and **outlierIQR** are independent of λ and thus, reusable (*n* = 1). The training in block 4 is partially λ -dependent (*n* = 2). Similarly, we assign the storage level **MEMORY_AND_DISK** to block 2 and 3, and **MEMORY** (avoids spilling to disk) to 1 and 4, respectively.

5.3 Operator Ordering

Traditional backend-agnostic operator ordering is suboptimal for multi-backend systems. For a holistic inter-backend parallelism via asynchronous operators, we introduce a new operator ordering algorithm, **MAXPARALLELIZE** that aims to maximize opportunities for concurrent execution of local and

Algorithm 1 Operator linearization**Input:** Operator DAG D with root R **Output:** List of instructions L

```

1: if HASNOREMOTEOPS( $D$ ) then           // All local OPs
2:    $L = \text{DEPTHFIRST}(D)$              // Linearize depth-first
   // Step1: Identify OP chains & count operators
3:  $SRoots = \text{COLLECTSPROOTS}(D)$ 
4:  $GRoots = \text{COLLECTGPROOTS}(D)$ 
5: for each list  $Roots$  in  $SRoots, GRoots$  do
6:   for each root  $r$  in  $Roots$  do
7:     if  $r$  is Spark then
8:        $nSPOps[r] = \text{COUNTSPOPS}(r)$ 
9:     else if  $r$  is GPU then
10:       $nGPOps[r] = \text{COUNTGPOPS}(r)$ 
   // Step2: Sort and linearize remote jobs (longer jobs first)
11:  $Roots = SRoots + GRoots$ 
12:  $Roots = \text{SORTROOTSBYOPCOUNT}(SRoots, nSPOps, nGPOps)$ 
13: for each root  $r$  in  $Roots$  do
14:   if  $r$  is Spark then
15:      $\text{DEPTHFIRST}(r, nSPOps[r], L)$ 
16:   else if  $r$  is GPU then
17:      $\text{DEPTHFIRST}(r, nGPOps[r], L)$ 
   // Step3: Linearize the rest of the local OPs
18:  $\text{DEPTHFIRST}(R, L)$            // Place unvisited nodes of  $D$ 

```

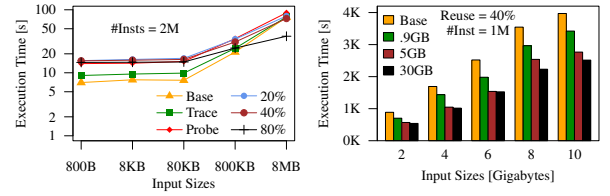
remote operator pipelines. As Algorithm 1 shows, we identify the operator chain roots (Spark action/prefetch/GPU-to-host copy) and linearize them in descending order of their lengths to maximize parallelism—longer operator chains allow for more concurrent execution, thus, increasing the degree of parallelism. The DEPTHFIRST method recursively processes sub-DAGs, with node memoization. This tight operator packing also improves memory usage by reducing the lifetime of dangling RDD references [42]. For Figure 9(a), MAXPARALLELIZE linearizes Job1 followed by Job2. The prefetch operators trigger these jobs concurrently, where solve and transpose wait on the future objects for results.

5.4 Applicability and Future Work

We now summarize the larger scopes, limitations, and future work of MEMPHIS and its components.

Applicability: Our proposed techniques generalize across diverse data systems with varying scope. (1) MEMPHIS applies to compilation-based, multi-backend systems for unified data processing [12, 31], integrated data analysis [21], polyglot, and AutoML. (2) *Fine-grained lineage tracing* applies broadly to systems with heterogeneous compilation and execution strategies, including ML libraries and databases [54], however lineage deduplication (discussed in LIMA [71]) requires control-flow access that is limited in Python-hosted libraries. (3) *Fine-grained reuse* is most effective in compilation-based ML systems [57] and relational databases [5]. (4) *Memory management and reuse in Spark* generalize to lazily-evaluated distributed frameworks like Dask [74] and Ray [58]. (5) *Reuse and memory management in GPU* naturally extend to multi-GPU and other accelerators. (6) Our *compiler optimizations*—including prefetch, operator ordering, eviction, and checkpoint injection—also extend to compilation-based systems and standalone key-value stores [25]. Overall, our runtime and compiler techniques extend across workloads and system architectures.

Limitations and Future Work: Our lineage-based reuse framework makes several tradeoffs and opens opportunities for future work. (1) *Reuse potential* is workload-dependent, however, even without reuse, our lineage tracing improves in-



(a) Reuse Overhead - Sizes (b) Cache Size Comparison

Figure 11: Reuse Overhead & Cache Size Influence.

terpretability and debuggability with negligible overhead. A future extension is enabling query processing over serialized lineage traces to analyze convergence and execution behavior. (2) Our cache is designed for process-wide sharing and requires extensions for *reusing across processes*. In addition, Python ML libraries may require instrumentation for tracing lineage [32]. (3) Despite handling non-determinism, reconstructed programs may yield slightly *different results* under varying configurations or versions. (4) Finally, a promising future work is a *cost-based, reuse-aware optimizer* that jointly optimizes query plan and operator scheduling.

6. EXPERIMENTS

We present some highlights from our experimental study [69]. Overall, our evaluation systematically scales data size, task complexity, and instruction counts across diverse ML pipelines—including data pre-processing, hyper-parameter optimization, matrix factorization, and DNN—to show benefits of multi-backend reuse in realistic settings.

Experimental Setting: We ran all experiments on a cluster of 8+1 scale-out nodes and a scale-up node with two NVIDIA A40 GPUs with 48 GB. The software stack comprises Ubuntu 20.04.6, Hadoop 3.3, Spark 3.5, and CUDA 10.2. The Spark cluster uses 38GB driver memory and 230GB executor memory; operators exceeding 7GB are compiled to Spark. The lineage cache is configured to 5GB on the driver and 55GB per executor. We compare MEMPHIS with different SystemDS configurations, application-specific reuse frameworks such as LIMA [71] (fine-grained reuse), HELIX [87] (coarse-grained reuse), CoordDL [55] (*input data pipeline* reuse), Clipper [19] (prediction reuse), and VISTA [60] (reuse in transfer learning), and PyTorch [64] as a strong baseline for DNN workloads on GPUs.

Reuse Overhead: We analyze the overhead of lineage tracing and cache probing using a hyper-parameter tuning workload based on L2SVM, varying input sizes (800B–8 MB), and reuse rates. With a fixed 200 iterations (2M instructions), increasing input size scales compute cost per instruction while tracing and cache management overheads remain constant; reuse is simulated by randomly repeating hyper-parameters, yielding 20–80% reusable operations. Figure 11(a) compares *Base* (vanilla SystemDS) with *Trace* (tracing only) and *Probe* (reuse enabled but no hits). For small inputs, execution time is dominated by interpretation and bookkeeping overhead, with tracing and probing increasing runtime by 1.3x and 2x, and reuse providing no benefit. For larger inputs (8 MB), these overheads become negligible, and reuse yields speedups from 1.1x (20%) to 3x (80%).

Cache Size Comparison: We evaluate driver cache sizes using the same workload with larger inputs (2–10 GB), 1M instructions, and 40% reuse, comparing cache sizes of 900 MB,

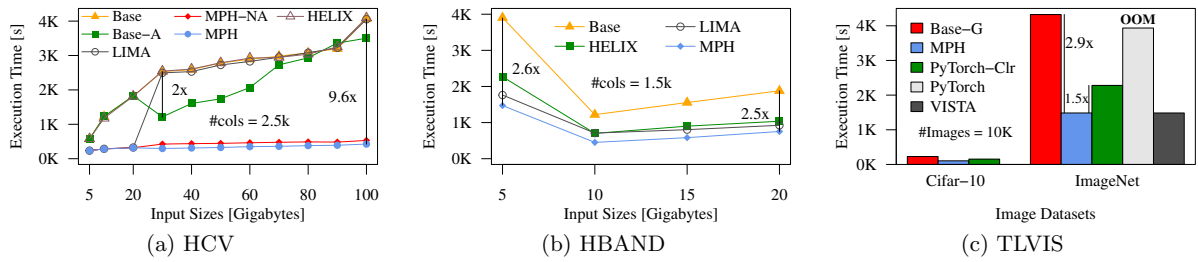


Figure 12: Performance of End-to-end ML Pipelines.

5 GB, and 30 GB. Inputs of 8 GB and 10 GB trigger Spark operations. Figure 11(b) shows even a 900 MB cache consistently achieves 1.2x speedup. While 5 GB and 30 GB caches perform similarly for smaller inputs, the larger cache yields slightly higher speedups for large inputs (1.6x vs. 1.4x). These gains stem from reuse of local intermediates, Spark RDDs and actions (including `prefetch`), proving the robustness of our eviction policies even under repeated evictions.

Hyper-parameter Tuning of Linear Regression (HCV): We evaluate multi-backend reuse on an end-to-end hyper-parameter tuning pipeline that performs cross-validated linear regression (Example 1) on synthetic datasets with 10 regularization parameters, selecting the best model using R^2 . We vary input sizes from 5-100GB and compare Base, LIMA, HELIX, and MEMPHIS (MPH), where Base-A and MPH enable asynchronous operators and Base and MPH-NA do not. As shown in Figure 12(a), MPH achieves up to 9.6x speedup over Base by reusing $\mathbf{X}^T \mathbf{X}$ and $\mathbf{X}^T \mathbf{y}$ across folds and overlapping execution via `prefetch`. From 25 GB onward, these intermediates are placed in Spark; LIMA reuses only local intermediates up to 20 GB, while MPH reuses in all settings. HELIX performs similarly to Base due to the lack of fine-grained reuse, Base-A achieves 2x speedup from concurrency for smaller inputs, and MPH is about 20% faster than MPH-NA due to parallel operator execution.

Hyperband Model Selection (HBAND): This experiment evaluates model search using a Hyperband-like multi-armed bandit [45] with weighted ensemble learning on synthetic datasets. HBAND first applies successive halving to tune hyper-parameters of L2SVM and multi-class logistic regression (MLRG) via a grid search over regularization (`reg`) and intercept options, iterating over five brackets that halve the `reg` list (starting from 25 values) while doubling iterations (from 10). It then applies weighted ensemble learning to combine the best models, optimizing ensemble weights through a random search over 1K configurations. As shown in Figure 12(b), MEMPHIS (MPH) yields 2.6x/2.5x speedups for 5 GB/20 GB inputs over Base by reusing successive-halving iterations and $\mathbf{XB}$ computations in class probability estimation, reusing approximately 4K RDDs, 2K Spark actions, and 40K local matrices. Our compiler enabled *delayed caching* of RDDs for MPH. Overall, MPH outperforms HELIX and LIMA by about 40%, and remains 20% faster than LIMA even for local operations.

Transfer Learning for Vision Models (TLVIS): Transfer learning is a widely used alternative to training large models from scratch. We evaluate MEMPHIS for transfer learning, where practitioners test multiple pre-trained models to identify the most suitable model-layer pair for a downstream task [60, 73]. TLVIS uses AlexNet [40], VGG16 [80],

and ResNet18 [34], extracting frozen feature layers ranging from Conv4-FC7 (AlexNet), Conv5-FC7 (VGG), and the last four residual blocks (ResNet), and ranks extracted features using a linear classifier proxy [63, 82]. Experiments run on CIFAR-10 and ImageNet test sets (10K images each) on GPUs. As shown in Figure 12(c), MEMPHIS achieves 2x and 3x speedups on CIFAR-10 and ImageNet by reusing intermediates across repeated feature extractions and efficient pointer recycling without degrading reuse (e.g., reusing and recycling 30K and 17.5K pointers for ImageNet). MEMPHIS compiles an `evict(100)` instruction between models (*eviction injection*) to free up the allocated pointers. VISTA performs similarly by applying CSE across pipelines. Compared to PyTorch, which uses frozen pre-trained models and GPU pre-loading, `torch.compile` suffers from out-of-memory errors and requires manual cache cleanup (`empty_cache()`) after each model (PyTorch-Clr). PyTorch—as a specialized system for DNNs—is 1.9x faster than Base-G (SystemDS-GPU) for TLVIS but 1.5x slower than MEMPHIS due to lack of reuse of repeated feature extractions.

7. CONCLUSIONS

After five years of developing this framework [68], we draw several key conclusions. As systems become increasingly heterogeneous across runtimes and backends, and ML workloads expand to multi-modal pipelines, LLMs, and AI agents, redundancy is both inevitable and progressively harder to manage. The diversity of workloads, execution models, and backends makes manual redundancy elimination infeasible and static cache management ineffective. We introduced MEMPHIS as a holistic framework for efficient, multi-backend reuse of intermediates and memory management. Our hierarchical lineage cache seamlessly allows reuse across heterogeneous backends and system architectures. We devised tailored eviction policies and memory management techniques for Spark and GPUs. Our compiler extensions improve reuse potential, reduce runtime overheads, and enable concurrent execution. Our experiments have shown robust improvements across diverse workloads. Overall, MEMPHIS eliminates redundancy in a principled and systematic manner while remaining broadly applicable to future workloads, backends, and evolving system architectures.

8. ACKNOWLEDGMENTS

We gratefully acknowledge funding from the German Federal Ministry of Research, Technology and Space (under research grant BIFOLD25B).

9. REFERENCES

- [1] Martín Abadi et al. TensorFlow: A System for Large-Scale Machine Learning. In *OSDI*, pages 265–283, 2016.
- [2] Christopher R. Aberger, Andrew Lamb, Kunle Olukotun, and Christopher Ré. Mind the Gap: Bridging Multi-Domain Query Workloads with EmptyHeaded. *PVLDB*, 10(12):1849–1852, 2017.
- [3] Divy Agrawal et al. RHEEM: enabling cross-platform data processing - may the big data be with you! -. *PVLDB*, 11(11):1414–1427, 2018.
- [4] Hani Al-Sayeh, Bunjamin Memishi, Muhammad Attahir Jibril, Marcus Paradies, and Kai-Uwe Sattler. Juggler: Autonomous cost optimization and performance prediction of big data applications. In *SIGMOD*, pages 1840–1854, 2022.
- [5] Alex Hall. Caching & reuse of subresults across queries, 2024. URL: <https://www.firebolt.io/blog/caching-reuse-of-subresults-across-queries>.
- [6] Alexander Alexandrov, Andreas Kunft, Asterios Katsifodimos, Felix Schöler, Lauritz Thamsen, Odej Kao, Tobias Herb, and Volker Markl. Implicit parallelism through deep language embedding. In *SIGMOD*, pages 47–61, 2015.
- [7] Antreas Antoniou, Amos J. Storkey, and Harrison Edwards. Augmenting image classifiers using data augmentation generative adversarial networks. In *ICANN*, volume 11141 of *Lecture Notes in Computer Science*, pages 594–603, 2018. doi:10.1007/978-3-030-01424-7_58.
- [8] Paul Barham et al. Pathways: Asynchronous distributed dataflow for ML. In Diana Marculescu, Yuejie Chi, and Carole-Jean Wu, editors, *MLSys*, 2022.
- [9] Nirvik Baruah, Peter Kraft, Fiodar Kazhamiaka, Peter Bailis, and Matei Zaharia. Parallelism-optimizing data placement for faster data-parallel computations. *PVLDB*, 16(4):760–771, 2022. doi:10.14778/3574245.3574260.
- [10] Sebastian Baunsgaard et al. Exdra: Exploratory data science on federated raw data. In *SIGMOD*, pages 2450–2463, 2021. doi:10.1145/3448016.3457549.
- [11] Denis Baylor et al. TFX: A tensorflow-based production-scale machine learning platform. In *KDD*, pages 1387–1395, 2017.
- [12] Kaustubh Beedkar, Bertty Contreras-Rojas, Haralampos Gavrilidis, Zoi Kaoudi, Volker Markl, Rodrigo Pardo-Meza, and Jorge-Arnulfo Quiané-Ruiz. Apache wayang: A unified data analytics framework. *SIGMOD Rec.*, 52(3):30–35, 2023.
- [13] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. Julia: A fresh approach to numerical computing. *SIAM Rev.*, 59(1):65–98, 2017.
- [14] Matthias Boehm et al. Systemml: Declarative machine learning on spark. *PVLDB*, 9(13):1425–1436, 2016. doi:10.14778/3007263.3007279.
- [15] Matthias Boehm et al. SystemDS: A Declarative Machine Learning System for the End-to-End Data Science Lifecycle. In *CIDR*, 2020. URL: <http://cidrdb.org/cidr2020/papers/p22-boehm-cidr20.pdf>.
- [16] Matthias Boehm, Arun Kumar, and Jun Yang. *Data Management in Machine Learning Systems*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2019.
- [17] Dami Choi and George Dahl. Speeding up neural network training with data echoing, 2020. URL: <https://ai.googleblog.com/2020/05/speeding-up-neural-network-training.html>.
- [18] PyTorch Contributors. Memory management of pytorch, 2023. URL: <https://pytorch.org/docs/stable/notes/cuda.html#memory-management>.
- [19] Daniel Crankshaw, Xin Wang, Giulio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. Clipper: A low-latency online prediction serving system. In *NSDI*, pages 613–627, 2017.
- [20] Ekin D. Cubuk, Barret Zoph, Dandelion Mané, Vijay Vasudevan, and Quoc V. Le. Autoaugment: Learning augmentation strategies from data. In *CVPR*, pages 113–123, 2019. doi:10.1109/CVPR.2019.00020.
- [21] Patrick Damme et al. DAPHNE: an open and extensible system infrastructure for integrated data analysis pipelines. In *CIDR*, 2022.
- [22] Piali Das et al. Amazon sagemaker autopilot: a white box automl solution at scale. In Sebastian Schelter, Steven Whang, and Julia Stoyanovich, editors, *DEEM@SIGMOD*, pages 2:1–2:7, 2020.
- [23] Behrouz Derakhshan, Alireza Rezaei Mahdiraji, Ziawash Abedjan, Tilmann Rabl, and Volker Markl. Optimizing Machine Learning Workloads in Collaborative Environments. In *SIGMOD*, pages 1701–1716, 2020. doi:10.1145/3318464.3389715.
- [24] Apache Beam Developers. The Unified Apache Beam Model, 2024. URL: <https://beam.apache.org/>.
- [25] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. Rocksdb: Evolution of development priorities in a key-value store serving large-scale applications. *ACM Trans. Storage*, 17(4):26:1–26:32, 2021. doi:10.1145/3483840.
- [26] Shiqing Fan et al. DAPPLE: a pipelined data parallel approach for training large models. In *PPoPP*, pages 431–445, 2021. doi:10.1145/3437801.3441593.
- [27] Haoyang Fang et al. Mlzero: A multi-agent system for end-to-end machine learning automation. In *Advances in Neural Information Processing Systems*, 2025. URL: <https://neurips.cc/virtual/2025/poster/119504>.
- [28] Matthias Feurer, Aaron Klein, Katharina Eggenberger, Jost Tobias Springenberg, Manuel Blum, and Frank Hutter. Auto-sklearn: Efficient and robust automated machine learning. In *Automated Machine Learning - Methods, Systems, Challenges*, pages 113–134. 2019.
- [29] Haralampos Gavrilidis, Kaustubh Beedkar, Jorge-Arnulfo Quiané-Ruiz, and Volker Markl. In-situ cross-database query processing. In *ICDE*, pages 2794–2807, 2023.
- [30] Victor Giannakouris, Nikolaos Papailiou, Dimitrios Tsoumakos, and Nectarios Koziris. Musql: Distributed SQL query execution over multiple engine environments. In *IEEE BigData*, pages 452–461, 2016.
- [31] Ionel Gog, Malte Schwarzkopf, Natacha Crooks, Matthew P. Grosvenor, Allen Clement, and Steven Hand. Musketeer: all for one, one for all in data processing systems. In *EuroSys*, pages 2:1–2:16, 2015.
- [32] Stefan Grafberger, Paul Groth, and Sebastian Schelter. Automating and optimizing data-centric

- what-if analyses on native machine learning pipelines. *Proc. ACM Manag. Data*, 1(2):128:1–128:26, 2023. doi:10.1145/3589273.
- [33] Dan Graur, Damien Aymon, Dan Kluser, Tanguy Albrici, Chandramohan A. Thekkath, and Ana Klimovic. Cachew: Machine learning input data processing as a service. In *ATC*, pages 689–706, 2022.
 - [34] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, pages 770–778, 2016.
 - [35] Mark Hildebrand, Jawad Khan, Sanjeev Trika, Jason Lowe-Power, and Venkatesh Akella. Autotm: Automatic tensor movement in heterogeneous memory systems using integer linear programming. In *ASPLOS*, pages 875–890, 2020.
 - [36] Milena Ivanova, Martin L. Kersten, Niels J. Nes, and Romulo Goncalves. An architecture for recycling intermediates in a column-store. In *SIGMOD*, pages 309–320, 2009. URL: <https://doi.org/10.1145/1559845.1559879>.
 - [37] Paras Jain et al. Checkmate: Breaking the memory wall with optimal tensor rematerialization. In *MLSys*, 2020.
 - [38] Abhinav Jangda et al. Breaking the computation and communication abstraction barrier in distributed machine learning workloads. In *ASPLOS*, pages 402–416, 2022. doi:10.1145/3503222.3507778.
 - [39] Antonios Kontaxakis, Dimitris Sacharidis, Alkis Simitsis, Alberto Abelló, and Sergi Nadal. HYPPO: Using Equivalences to Optimize Pipelines in Exploratory Machine Learning. In *ICDE*, pages 221–234, 2024.
 - [40] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *NeurIPS*, pages 1106–1114, 2012.
 - [41] Fan Lai, Xiangfeng Zhu, Harsha V. Madhyastha, and Mosharaf Chowdhury. Oort: Efficient federated learning via guided participant selection. In *OSDI*, pages 19–35, 2021.
 - [42] Lukas Landgraf, Wolfgang Lehner, Florian Wolf, and Alexander Boehm. Memory efficient scheduling of query pipeline execution. In *CIDR*, 2022.
 - [43] Yunseong Lee, Alberto Scolari, Byung-Gon Chun, Marco Domenico Santambrogio, Markus Weimer, and Matteo Interlandi. PRETZEL: opening the black box of machine learning prediction serving systems. In *OSDI*, pages 611–626, 2018.
 - [44] Liam Li, Kevin G. Jamieson, Afshin Rostamizadeh, Ekaterina Gonina, Jonathan Ben-tzur, Moritz Hardt, Benjamin Recht, and Ameet Talwalkar. A system for massively parallel hyperparameter tuning. In *MLSys*, 2020.
 - [45] Lisha Li, Kevin G. Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *J. Mach. Learn. Res.*, 18:185:1–185:52, 2017. URL: <http://jmlr.org/papers/v18/16-558.html>.
 - [46] Edo Liberty et al. Elastic machine learning algorithms in amazon sagemaker. In *SIGMOD*, pages 731–737, 2020. doi:10.1145/3318464.3386126.
 - [47] Chao Liu, Hung-chih Yang, Jinliang Fan, Li-wei He, and Yi-Min Wang. Distributed nonnegative matrix factorization for web-scale dyadic data analysis on mapreduce. In *WWW*, pages 681–690, 2010.
 - [48] Scott M. Lundberg and Su-In Lee. A unified approach to interpreting model predictions. In *NIPS*, pages 4765–4774, 2017.
 - [49] Hector Mendoza, Aaron Klein, Matthias Feurer, Jost Tobias Springenberg, and Frank Hutter. Towards automatically-tuned neural networks. In *ICML*, volume 64, pages 58–65, 2016.
 - [50] Chen Meng, Minmin Sun, Jun Yang, Minghui Qiu, and Yang Gu. Training deeper models by gpu memory optimization on tensorflow. In *ML System@NeurIPS Workshop*, 2017.
 - [51] Xiangrui Meng et al. MLlib: Machine Learning in Apache Spark. *JMLR*, 17:34:1–34:7, 2016.
 - [52] Fraser Mince, Dzung Dinh, Jonas Kgomo, Neil Thompson, and Sara Hooker. The grand illusion: The myth of software portability and implications for ML progress. In *NeurIPS 2023*, 2023.
 - [53] Azalia Mirhoseini et al. Device placement optimization with reinforcement learning. In Doina Precup and Yee Whye Teh, editors, *ICML*, pages 2430–2439, 2017.
 - [54] Haneen Mohammed and Eugene Wu. Lineage capture trade-offs: A case study in duckdb. In *ProvenanceWeek@SIGMOD*, pages 32–36, 2025.
 - [55] Jayashree Mohan, Amar Phanishayee, Ashish Raniwala, and Vijay Chidambaram. Analyzing and mitigating data stalls in DNN training. *PVLDB*, 14(5):771–784, 2021. doi:10.14778/3446095.3446100.
 - [56] Hubert Mohr-Daurat, Xuan Sun, and Holger Pirk. BOSS - an architecture for database kernel composition. *PVLDB*, 17(4):877–890, 2023.
 - [57] Dan Moldovan, James M. Decker, Fei Wang, Andrew A. Johnson, Brian K. Lee, Zachary Nado, D. Sculley, Tiark Rompf, and Alexander B. Wiltschko. Autograph: Imperative-style coding with graph-based performance. In *MLSys*, 2019. URL: <https://proceedings.mlsys.org/book/272.pdf>.
 - [58] Philipp Moritz et al. Ray: A distributed framework for emerging AI applications. In *OSDI*, pages 561–577, 2018.
 - [59] Derek Gordon Murray, Jiri Simsa, Ana Klimovic, and Ihor Indyk. tf.data: A machine learning data processing framework. *PVLDB*, 14(12):2945–2958, 2021. doi:10.14778/3476311.3476374.
 - [60] Supun Nakandala and Arun Kumar. Vista: Optimized system for declarative feature transfer from deep cnns at scale. In *SIGMOD*, pages 1685–1700, 2020.
 - [61] Jaehyun Nam, Jinsung Yoon, Jiefeng Chen, Jinwoo Shin, Serkan Ö. Arik, and Tomas Pfister. MLE-STAR: machine learning engineering agent via search and targeted refinement. In *NeurIPS*, 2025.
 - [62] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. Pipedream: generalized pipeline parallelism for DNN training. In *SOSP*, pages 1–15, 2019.
 - [63] Cuong V. Nguyen, Tal Hassner, Matthias W. Seeger, and Cédric Archambeau. LEEP: A new measure to evaluate transferability of learned representations. In *ICML*, pages 7294–7305, 2020.

- [64] Adam Paszke et al. Pytorch: An imperative style, high-performance deep learning library. In *NeurIPS*, pages 8024–8035, 2019.
- [65] Fabian Pedregosa et al. Scikit-learn: Machine learning in python. *JMLR*, 12:2825–2830, 2011.
- [66] Xuan Peng, Xuanhua Shi, Hulin Dai, Hai Jin, Weiliang Ma, Qian Xiong, Fan Yang, and Xuehai Qian. Capuchin: Tensor-based GPU memory management for deep learning. In *ASPLOS*, pages 891–905, 2020. doi:10.1145/3373376.3378505.
- [67] Tiago B. G. Perez, Xiaobo Zhou, and Dazhao Cheng. Reference-distance eviction and prefetching for cache management in spark. In *ICPP*, pages 88:1–88:10, 2018.
- [68] Arnab Phani. *Fine-grained reuse and feature transformations in machine learning systems*. PhD thesis, Technical University of Berlin, Germany, 2025. URL: <https://nbn-resolving.org/urn:nbn:de:101:1-2504300159545.425811098571>.
- [69] Arnab Phani and Matthias Boehm. MEMPHIS: holistic lineage-based reuse and memory management for multi-backend ML systems. In *EDBT*, pages 255–269. OpenProceedings.org, 2025. doi:10.48786/EDBT.2025.21.
- [70] Arnab Phani, Lukas Erlbacher, and Matthias Boehm. UPLIFT: parallelization strategies for feature transformations in machine learning workloads. *PVLDB*, 15(11):2929–2938, 2022.
- [71] Arnab Phani, Benjamin Rath, and Matthias Boehm. LIMA: fine-grained lineage tracing and reuse in machine learning systems. In *SIGMOD*, pages 1426–1439, 2021. doi:10.1145/3448016.3452788.
- [72] Sergey Redyuk, Zoi Kaoudi, Volker Markl, and Sebastian Schelter. Automating data quality validation for dynamic data ingestion. In *EDBT*, pages 61–72, 2021. doi:10.5441/002/EDBT.2021.07.
- [73] Cédric Renggli, Xiaozhe Yao, Luka Kolar, Luka Rimanic, Ana Klimovic, and Ce Zhang. Shift: An efficient, flexible search engine for transfer learning. *PVLDB*, 16(2):304–316, 2022.
- [74] Matthew Rocklin. Dask: Parallel computation with blocked algorithms and task scheduling. In *SciPy*, pages 130 – 136, 2015.
- [75] Svetlana Sagadeeva and Matthias Boehm. Sliceline: Fast, linear-algebra-based slice finding for ML model debugging. In *SIGMOD*, pages 2290–2299, 2021.
- [76] Sebastian Schelter, Dustin Lange, Philipp Schmidt, Meltem Celikel, Felix Bießmann, and Andreas Grafberger. Automating Large-Scale Data Quality Verification. *PVLDB*, 11(12):1781–1794, 2018.
- [77] Noam Shazeer et al. Mesh-tensorflow: Deep learning for supercomputers. In *NeurIPS*, pages 10435–10444, 2018.
- [78] Shafaq Siddiqi, Roman Kern, and Matthias Boehm. Saga: A scalable framework for optimizing data cleaning pipelines for machine learning applications. *Proc. ACM Manag. Data*, 2024.
- [79] Shafaq Siddiqi, Arnab Phani, Roman Kern, and Matthias Boehm. Saga++: A scalable framework for optimizing data cleaning pipelines for machine learning applications. *ACM TODS*, 2025.
- [80] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *ICLR*, 2015.
- [81] Evan R. Sparks, Shivaram Venkataraman, Tomer Kaftan, Michael J. Franklin, and Benjamin Recht. Keystoneml: Optimizing pipelines for large-scale advanced analytics. In *ICDE*, pages 535–546, 2017.
- [82] Anh Tuan Tran, Cuong V. Nguyen, and Tal Hassner. Transferability and hardness of supervised classification tasks. In *ICCV*, pages 1395–1405, 2019.
- [83] Taegeon Um, Byungsoo Oh, Byeongchan Seo, Minhyeok Kweun, Goeun Kim, and Woo-Yeon Lee. Fastflow: Accelerating deep learning model training with smart offloading of input data pipeline. *PVLDB*, 16(5):1086–1099, 2023.
- [84] Stéfan van der Walt, S. Chris Colbert, and Gaël Varoquaux. The numpy array: A structure for efficient numerical computation. *Comput. Sci. Eng.*, 13(2):22–30, 2011.
- [85] Manasi Vartak, Joana M. F. da Trindade, Samuel Madden, and Matei Zaharia. MISTIQUE: A system to store and query model intermediates for model diagnosis. In *SIGMOD*, pages 1285–1300, 2018.
- [86] Yongji Wu, Matthew Lentz, Danyang Zhuo, and Yao Lu. Serving and optimizing machine learning workflows on heterogeneous infrastructures. *PVLDB*, 16(3):406–419, 2022. doi:10.14778/3570690.3570692.
- [87] Doris Xin, Stephen Macke, Litian Ma, Jialin Liu, Shuchen Song, and Aditya G. Parameswaran. Helix: Holistic optimization for accelerating iterative machine learning. *PVLDB*, 12(4):446–460, 2018.
- [88] Luna Xu, Min Li, Li Zhang, Ali Raza Butt, Yandong Wang, and Zane Zhenhua Hu. MEMENTO: dynamic memory management for in-memory data analytic platforms. In *IPDPS*, pages 383–392, 2016.
- [89] Matei Zaharia et al. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *NSDI*, pages 15–28, 2012.
- [90] Ce Zhang, Arun Kumar, and Christopher Ré. Materialization optimizations for feature selection workloads. In *SIGMOD*, pages 265–276, 2014.
- [91] Yanli Zhao et al. Pytorch FSDP: experiences on scaling fully sharded data parallel. *PVLDB*, 16(12):3848–3860, 2023.
- [92] Lianmin Zheng et al. Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning. In *OSDI*, pages 559–578, 2022.
- [93] Jingren Zhou, Per-Åke Larson, Johann Christoph Freytag, and Wolfgang Lehner. Efficient exploitation of similar subexpressions for query processing. In *SIGMOD*, pages 533–544, 2007. URL: <https://doi.org/10.1145/1247480.1247540>.

Technical Perspective: Rel

Viktor Leis, Thomas Neumann
TUM
{leis|neumann}@in.tum.de

For more than half a century, relational data management has lived in a “two-language world”: a powerful declarative sublanguage for data (SQL) glued to a host programming language for everything else. This split into two worlds creates friction: First, there is an “impedance mismatch” between the set oriented query language SQL and the usually more imperative host programming language. And second, while programming languages aim for producing reusable components, SQL queries are often ad-hoc, and there is little if any reuse of logic and functionality across queries.

The Rel language makes a clean and ambitious break with that paradigm. Rather than treating relational querying as a bolt-on, Rel is designed as a full programming language whose central abstraction is the relation itself. The paper goes beyond syntactic polish or yet another query notation, targeting both weaknesses of the traditional split: First, complex logic can be expressed naturally on relational data, eliminating the impedance mismatch between host language and query language. And second, the language allows for higher order constructs, which is the basis for implementing libraries of reusable components.

This shift directly targets long-recognized pain points of SQL. As we discuss in our CIDR’24 critique of SQL [2], it lacks extensibility and abstraction, forcing new capabilities to arrive as ever more surface syntax instead of reusable language-level constructs. Rel responds by elevating relational programming beyond “querying”: it begins from Datalog and first-order logic, embraces recursion, and introduces abstraction and application as key constructs, enabling users to succinctly express computations that routinely escape the traditional boundaries of SQL. The result is not SQL plus recursion, but a language in which relations play the role that functions play in functional languages or procedures in imperative ones—a principled reframing that opens the door to richer, compositional program structure.

A particularly elegant idea behind Rel is its “growing a language” philosophy: the paper explicitly prioritizes mechanisms that let users build the language outward through libraries, rather than centralizing power in an ever-expanding standard. Rel’s authors emphasize programming-in-the-large features precisely because large-scale development depends on modularity, parameterization, and reuse—not on

an endless accumulation of special cases.

In the paper’s organization, the progression from Rel’s core to “Enabling Programming in the Large” and then to “Growing the Language” underscores a design that treats libraries as first-class citizens. Concretely, Rel supports passing richer parameters than individual scalar values (tuple and relation variables) and provides constructs that let developers package reusable relational building blocks that remain uniform with the rest of the language. This is exactly the kind of extensibility SQL struggles to offer.

Equally important for a language proposal is that Rel comes with formal semantics. The paper includes an addendum giving a semantics for a core of Rel, spelling out the data model, environments, and the semantics of expressions, formulas, and programs. A semantics-first foundation enables rigorous reasoning about program meaning, optimization opportunities, and equivalence. In that sense, Rel aligns with recent calls for semantics-first relational representations (e.g., ARQL/ARC [1]) that separate intent from surface syntax and make relational structure explicit.

In sum, Rel is based on a bold idea: that relational thinking can be the primary programming model, not a specialized fragment embedded in something else. By (1) treating Rel as a programming language capable of expressing rich computations beyond traditional SQL querying, (2) enabling a library ecosystem that lets users grow the language rather than wait for standards to ossify, and (3) grounding the whole design in formal semantics, this paper charts a compelling path toward simpler architectures, more reusable data logic, and more principled relational systems.

1. REFERENCES

- [1] W. Gatterbauer and D. M. Sabale. Database research needs an abstract relational query language. In *CIDR*, 2026.
- [2] T. Neumann and V. Leis. A critique of modern SQL and a proposal towards a simple and expressive query language. In *CIDR*, 2024.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 2024 ACM 0001-0782/24/0X00 ...\$5.00.

Relational Programming in Rel

Molham Aref¹, Paolo Guagliardo², George Kastrinis¹, Leonid Libkin^{1,2,7},
Victor Marsault³, Wim Martens^{1,4}, Mary McGrath¹, Filip Murlak⁵, Nathaniel Nystrom¹,
Liat Peterfreund⁶, Allison Rogers¹, Cristina Sirangelo^{1,7},
Domagoj Vrgoč⁸, David Zhao¹, Abdul Zreika¹

¹RelationalAI ²University of Edinburgh ³LIGM, Univ. Gustave Eiffel, CNRS ⁴University of Bayreuth
⁵University of Warsaw ⁶Hebrew University ⁷IRIF, Université Paris Cité, CNRS ⁸PUC Chile

ABSTRACT

From the moment of their inception, languages for relational data have been described as *sublanguages* embedded in a host programming language. Rel is a new relational language that goes beyond this paradigm, with features that allow for programming in the large, making it possible to fully describe end to end application semantics. With the new approach we can model the semantics of entire enterprise applications relationally, which helps significantly reduce architecture complexity and avoid the well-known *impedance mismatch* problem. This paradigm shift is enabled by 50 years of database research, making it possible to revisit the sublanguage/host language paradigm, starting from the fundamental principles. We present a gentle introduction to Rel and the principles behind its design philosophy.

1. INTRODUCTION

Languages for relational data have always been described as *sublanguages* [9, 10]. This means that such languages are not designed to describe the entire end-to-end application logic of programs that involve data, but rather focus on specific operations that concern the storage, retrieval, or manipulation of data within such programs. This view of database languages made a lot of sense fifty years ago, when database query languages were being introduced in an environment of procedural imperative programming, when declarative languages were a revolutionary idea, and when it was unclear if this idea could be extended beyond its domain-specific use. Today, SQL is the prime success story of a declarative and domain-specific language. It remains in high demand by tech employers [7], which is a remarkable achievement for a 40-year-old programming (sub)language.

The sublanguage paradigm however has important limitations. To start with, it entails having to communicate with a host programming language. This causes the well-known *impedance mismatch*: the two languages have different data types, different data structures, different memory models, etc. Often they are even based on *different programming paradigms*, one being fundamentally declarative and the other fundamentally imperative. Furthermore, the two languages typically have separate runtime

environments, which limits the optimizations that can be applied to programs that use both languages. The host language runtime environment may not support features of the database such as automatic out of core computation, automatic parallelism, incremental computation. In other words, the impedance mismatch:

1. causes extra complexity for developers, unrelated to solving the problem at hand;
2. limits the capabilities of automatic support that we are used to in databases.

Another effect of the sublanguage paradigm is that query languages are not equipped with important features needed for programming in the large, as these are delegated to the host language. Most notably, query languages lack support for building libraries. Indeed, SQL does not have a standard library, and when new features are needed they are added by means of expanding the language itself. Over the years, this led to an 11-part ISO standard comprising well over 4000 pages which is an order of magnitude larger than the C standard. This breaks a fundamental principle of programming language design formulated by Steele [25]: *define a small core and provide the functionality to build libraries*.

Rel aims to go beyond the sublanguage paradigm. The database industry and research communities have accumulated a vast array of insights that reshuffle the cards on which the sublanguage design decision was based. We ask ourselves, can we *design and implement* a language that natively handles data and semantics in a database, preserving the bedrock principles of databases (such as the data independence, communicability, and set-at-a-time processing objectives [12]), providing the programmer with the necessary constructs to factor semantics out of application programs? Such a full-featured language for data and semantics would allow for new powerful simplifications and optimizations owing to a single runtime environment, increasing the productivity of users and application developers.

In this paper, we give a gentle introduction to Rel while presenting our approach to language design. With it, we aim to provide language support for enterprise AI that brings semantics close to data and drastically simplifies application development. We build upon the foundational ideas behind the relational model — data modeling via normalization and declarative programming — and take them further. On the data modeling front, we advocate for *full normalization* as a means of coupling data and semantics. On the language front, we advocate for extending the declarative paradigm to the entire language, beyond its data management fragment.

Rel has been designed and implemented based on those

Copyright is held by the owners/author(s). Publication rights licensed to the ACM. This is a minor revision of the paper “Rel: A Programming Language for Relational Data”, published in the 2025 International Conference on Management of Data (SIGMOD), <https://doi.org/10.1145/3722212.3724450>.

principles. Its key features are:

1. manipulation of logical formulas and entire relations;
2. powerful *recursion* built on the foundations of Datalog;
3. tools that enable writing libraries:
 - *abstraction* and *application* as key constructs;
 - variables that can *range over tuples and relations*.

We touch upon all these points in the course of the paper.

Rel in the Real World. Today, Rel is in use by several large enterprises to build end-to-end applications that include fraud detection, taxation, supply chain management, and others. The entire business logic for these applications is modeled in Rel, leveraging its support for programming in the large. Applications developed in Rel have run significantly faster and scaled to much larger data sets with drastically smaller code bases, when compared to the legacy applications that they replaced (up to 95% code reduction). We see this as a clear sign that the foundational concepts behind Rel deliver.

Models and Languages for Enterprise AI. Rel is aimed to program intelligent applications, leveraging a variety of AI techniques and reasoners to improve and automate decision making. The challenge, and the biggest cost here is to tame the highly complex mix of disparate technology stacks (e.g., for transactional, analytical, planning, graph, predictive, and prescriptive reasoning), each with its own data management methods and programming paradigm.

For example, assume that we want to answer the query “How can we price our items to maximize income from sales in western states in the next quarter?” Answering it comes with many challenges. First of all, it requires a combination of predictive reasoning (for predicting sales in the next quarter depending on pricing), solvers (to maximize income), and database querying. Second, the question requires precise semantics: how does the enterprise define profits, what is the date range for the next quarter, which states do we consider to be the western region? So, to express this query — whether by a human or by an agent — we need to provide the semantics of the concepts the query uses. Rel is designed to ease the burden of building such a semantic layer, and to ease writing programs that combine these different technologies. We come back to this in Section 6.4.

Furthermore, we strongly advocate for full normalization of data. Not only does it avoid semantically troublesome features such as nulls and duplicates, but it also represents data in the shape of a relational knowledge graph. This makes definitions required by the semantic layer particularly easy, and is a key reason why we advocate for the data-centric approach based on a declarative language and full normalization as a foundation for enterprise AI.

Design Principles of Rel. Codd [8, 9, 11, 12] introduced three main concepts: a data model based on n -ary relations, a normal form for database relations, and a universal (sub)language for querying database relations. We revisit these in varying depth: (1) we firmly stick to the tried and trusted relational data model, (2) we discuss data modeling and normal forms in Section 2, and (3) Rel, our proposed relational programming language, in Sections 3–6.

2. DATA MODELING

In this section, we explain the data modeling principles that we recommend and will help build robust applications.

Pure Relational Model. In the relational model, the n -tuples in each relation represent facts about the domain being modeled. Since a fact can neither hold multiple times, nor hold partially, the pure relational model [8] has neither nulls, nor multiplicities (bags), i.e., it is based on two-valued predicate logic and set semantics. We take this one step further: tuples represent facts that are *indivisible*. A complex fact like “Edgar was born on 19 August 1923 in Underhill” is better thought of as two indivisible facts, “Edgar was born on 19 August 1923” and “Edgar was born in Underhill”. Rel shares this perspective with fact-based modeling [18], inspired by the same work referenced by Codd [11]. As facts involve real-world concepts rather than database constants (*things*, *not strings*), tuples should store unique, context-independent representations of these concepts. In the above example, *Underhill* is the area on the Isle of Portland in Dorset (UK), not the town in Wisconsin nor the travel name of Frodo Baggins; the database should be able to distinguish between them all. The combination of indivisibility of facts and the *things*, *not strings* paradigm gives rise to Rel’s *graph normal form (GNF)*, as we explain below.

Graph Normal Form. GNF relies on a conceptual model that distinguishes between *entities* (products, orders, etc.) and *values* (integers, dates, etc.). In relations, values are represented by themselves as usual, whereas entities are represented by internal *identifiers*. In GNF, identifiers are disjoint from values, and every entity in the database is represented by an identifier that is unique within the entire database. So, GNF does not allow disjoint concepts, such as product and order, to have the same identifier in the database. We call this the *unique identifier property*.

In summary, *graph normal form (GNF)* comprises the following conditions:

- (1) for each k -ary relation R ,
 - all k columns of R are the key, or
 - the first $k - 1$ columns of R are the key for R ;
- (2) the unique identifier property holds.

Condition (1) captures indivisibility of facts by requiring each relation to be fully normalized. Incidentally, notice that this notion basically coincides with *sixth normal form (6NF)* [14]. On the other hand, condition (2) captures the *things*, *not strings* paradigm by providing each entity with its unique identifier. For example, information about people could be represented using a GNF schema with relations `person(p)`, `birthDate(p,d)`, `height(p,h)`, `mother(p,m)`, `father(p,f)`, `dog(x)`, `cat(x)`, `hasPet(p,x)`, where `person(p)` tells us that `p` is a person, etc. We always define predicates such that the ordering of their tuples is consistent with how the predicate is read: `birthDate(p,d)` means that the birth date of `p` is `d`; `mother(p,m)` means that the mother of `p` is `m`, etc. Note that in Rel attributes of relations are not associated with names: they are referred by their position only. In this example we use unary relations to identify entities, thus effectively enforcing the unique identifier property.

No nulls, no bags. Under GNF (and 6NF), relations *do not contain nulls* (instead of using a null in the non-key column, we simply omit the whole tuple), and are *sets* (keys cannot repeat). This brings us back to Codd’s original relational model; nulls and bags were introduced later for reasons that were pragmatic *at the time*, but caused problems further down the road. Nulls are known as one of the most problematic features of relational databases [13], with often unclear or misunderstood semantics [26], and a source

of many programming errors [6]. Recursion — essential for programming in the large — when coupled with bag semantics quickly leads to very large or even infinite multiplicities, and undecidability of query optimization [16, 19, 21]. Bag semantics is not needed for aggregation either, as we shall explain in Section 5. These problems disappear when we adopt “no nulls/no bags”, justifying it from the language design perspective. Furthermore, both set semantics and the absence of nulls restore classical logical equivalences [23, 22], thereby giving the query optimizer more headroom to do its work. Other benefits of GNF such as semantic stability and support for temporal features are discussed in [2].

3. INTRODUCTION TO REL

The starting point of Rel is Datalog with first-order formulas in rule bodies. We assume a basic level of familiarity with the relational data model [24, 3]. Throughout the paper, we will base our examples on the data in Figure 1, which is fully normalized, as we require in GNF (Section 2). The unique identifier property for people is enforced by the unary relation `person`, which (in the full schema) is disjoint from `dog` or `cat`. For the sake of readability, in the paper we do use strings to refer to these entities.

A Rel program is a set of rules, the most basic of which has the form

```
def RName( VariableList ) : RelExpression
```

A rule is structured like a Datalog rule: `RName` is a relation name, `VariableList` is a list of variables, and `RelExpression` is an expression that evaluates to a result that gives meaning to the variables in `VariableList`. For example, using the schema in Section 2, consider the rule

```
def parent(x,y) : mother(x,y) or father(x,y)
```

which defines `parent` as a new relation that is derived from `mother` and `father`, containing all pairs $\langle c, p \rangle$ such that p is the mother or father of c . For example, in the instance from Figure 1 we have that the tuple $\langle \text{“John”, “Emma”} \rangle$ belongs to the `mother` relation, while the tuple $\langle \text{“John”, “Joe”} \rangle$ belongs to the `father` relation, meaning that both tuples will be added to `parent`. Throughout, we write tuples using angular brackets (for example, $\langle \text{“John”} \rangle$ is a unary tuple, and $\langle \text{“John”, “Emma”} \rangle$ is a binary tuple).

Rule bodies can use full first-order formulas including both existential and universal quantification, as well as all Boolean operations. In the example

```
def sibling(x,y) :
  exists ((p) | parent(x,p) and parent(y,p) and x != y)
```

we say that x and y are siblings if they have a common parent and they are different. If we consider the database from Figure 1 we can see that “Kate” and “John” are siblings, since they have the same parent. Likewise, with the rule

```
def hasNoCats(p) : person(p) and
  forall ((x) | hasPet(p,x) implies not cat(x))
```

we say that p is in `hasNoCats` if p is a person and none of their pets are cats. Over the database of Figure 1 all people except “Emma” are returned, since one of her pets is a cat.

As in Datalog, rules can be recursive. For example, if we wish to express the ancestor relationship between people, we can use the following program consisting of two rules:

```
def ancestor(x,y) : parent(x,y)
def ancestor(x,y) :
  exists ((z) | parent(x,z) and ancestor(z,y))
```

The first rule states that each *parent* is also an *ancestor*, while the second rule states that each parent of an ancestor is also an ancestor. For instance, in the example of Figure 1 we have that $\langle \text{“Kate”, “Judith”} \rangle$, $\langle \text{“Kate”, “Anne”} \rangle$ and $\langle \text{“Kate”, “Emma”} \rangle$ all belong to `ancestor`, since the mother/parent of “Kate” is “Emma”, the mother of “Emma” is “Anne” and the mother of “Anne” is “Judith”.

Although the program above is fairly easy to understand, this is much less so for general recursive programs that involve negation in rule bodies. For this reason, Rel adopts partial fixpoint semantics (iterate until a fixpoint is reached), as we discuss in our paper [1].

Infinite Relations. Rel allows using and even defining infinite relations. For example, we already used the infinite relation `!=` (“is different from”) in the definition of `sibling`. Using another infinite relation `>=`, one can write

```
def youngestSibling(x) :
  forall ((y) | sibling(x,y) implies
    exists ((d1) | birthDate(x,d1) and
      exists ((d2) | birthDate(y,d2) and d1 >= d2)))
```

to say that x is a youngest sibling. Referring again to the database in Figure 1, we can see that $\langle \text{“John”} \rangle$ is in `youngestSibling`, since his only other sibling, “Kate”, is older. Note also that $\langle \text{“Emma”} \rangle$ is added to `youngestSibling` since she has no siblings and the universal quantification binds no y in the formula above. We will later see in detail how we can write this rule more conveniently as

```
def youngestSibling(x) :
  forall ((y) | sibling(x,y)
    implies birthDate[x] >= birthDate[y])
```

using *partial application*. For now, the reader can simply read `birthDate[x]` as “the birthdate of x ”.

Another natural use case for infinite relations is arithmetic. For instance, Rel uses `Int` to denote an infinite unary relation containing all integers and `add`, `multiply`, etc. for arithmetic. For example, `add(x,y,z)` means that the sum of x and y equals z , and similarly for the other arithmetic relations. These relations can also be written with the usual infix notation, so we can write

```
def addInverse(x,y) : x + y = 0
```

to define an (infinite) relation containing the pairs of numbers that are additive inverses of each other, or

```
def absolute(x,y) :
  (x >= 0 and y = x) or (x < 0 and y = -1 * x)
```

to define when the absolute value of x is y .

Note that the derived relations in the last two examples are infinite as well. In general, infinite relations not only arise from the use of primitive ones (such as `+` or `>` in the definition of `addInverse` and `absolute` above); they can also be obtained starting from finite relations, as in classical first-order logic, using, e.g., negation or disjunction. For example:

```
def nonHuman(x) : not person(x)
```

Infinite relations have an important role in Rel, as they allow the extension of the basic core language with libraries.

Safety. Being able to define infinite relations brings up so-called *safety* issues: to ensure that queries are computable, they should return only finitely many results.

Classical relational languages do not allow the definition of infinite relations to circumvent this problem. Relational calculus and SQL use *range restriction*. Specifically, variables can only range over elements in the database or those

person	mother		father		birthDate		height	
"Emma"	"John"	"Emma"	"John"	"Joe"	"John"	21-03-1988	"Kate"	187
"John"	"Kate"	"Emma"	"Kate"	"Joe"	"Kate"	28-06-1986	"Joe"	184
"Joe"	"Emma"	"Anne"			"Emma"	11-07-1957		
"Kate"	"Anne"	"Judith"						
"Anne"								
"Judith"								
				hasPet	dog	cat		
				"Emma"	"Snuffles"			
				"Joe"	"Luna"			
					"Charlie"			
					"Luna"			

Figure 1: Example database containing information about lineage. Some relations are omitted for brevity.

constructed from database entries by means of expressions, thereby rendering ranges of variables finite.

Rel takes a more flexible approach, as infinite relations are essential to the language. It allows unsafe definitions (i.e., unsafe rules within entire programs) as long as the final query is safe. For instance, the definition of `absolute` is not a safe query, because the relation is infinite. But if we add the definition `q` below, the query with output `q` is safe:

```
def q(x,y,h) :
  parent(x,y) and h=absolute[height[x] - height[y]]
```

The query outputs all triples $\langle x, y, h \rangle$ such that x is parent of y and h is (the absolute value of) their height difference.

Since safety is an undecidable condition [3], it is impossible to distinguish all safe and unsafe queries. The Rel engine takes a conservative approach, ensuring that it never attempts to evaluate an expression that could be unsafe. To this end, the engine reasons about safety of queries using a set of rules, based on [17].

4. PROGRAMMING IN THE LARGE

We now discuss language features that, together with the basics from Section 3, enable Rel to do programming in the large and, crucially, give it the power to do meta-programming and define libraries. For this, we extend the core of Rel with four fundamental ingredients:

- (1) tuple variables
- (2) relation variables
- (3) relational application, and
- (4) abstraction.

Tuple and relation variables are needed for enabling library writing with a small core language: these are mechanisms for passing parameters that are more complex than individual values. Relational application and abstraction are relational analogs of lambda-abstraction and application that are ubiquitous in programming languages. In our design, we build upon ideas of Boute [5] who showed how to incorporate these basic programming language building blocks into predicate logic, or relational calculus.

Tuple Variables. Assume that we want to compute the cross product (Cartesian product) of two binary relations `U` and `V`, that is, the set of tuples $\langle a, b, c, d \rangle$ such that $\langle a, b \rangle \in U$ and $\langle c, d \rangle \in V$. We can write this as:

```
def crossProductUV(a,b,c,d) : U(a,b) and V(c,d)
```

But what if `V` were ternary? We would have to write:

```
def crossProductUV(a,b,c,d,e) : U(a,b) and V(c,d,e)
```

Writing such code for all different arities is not only repetitive and error-prone, but also incomplete, because there is a maximal arity for which the code works. This is why Rel has *tuple variables*, which have trailing dots. Using tuple variables, we can define the cross product of `U` and `V` without knowing their arities:

```
def crossProductUV(x...,y...) : U(x...) and V(y...)
```

Here, we are saying that the cross product of `U` and `V` is the set of tuples $(x..., y...)$ such that $x...$ binds to a *tuple* in `U` and $y...$ binds to a *tuple* in `V`. We note that, in general, tuple variables do not need to bind to *entire* tuples; they can bind to arbitrary-length sub-tuples (even sub-tuples of length zero).

Relation Variables. Being able to define `ProductUV` independently of the arities of `U` and `V` is nice, but it would be even better if we could pass `U` and `V` as parameters to a more general operator: we would like

```
crossProduct[U,V]
```

to return the cross product of `U` and `V`. We will achieve this in two steps: the first is *relation variables*, the second is *relational application*.

In Rel syntax, relation variables in the head of rules are indicated by enclosing them in curly brackets:

```
def crossProduct({A},{B},x...,y...) : A(x...) and B(y...)
```

Then, we can write `crossProduct(U,V,a,b,c,d)` to test if $\langle a, b, c, d \rangle$ is in the cross product of `U` and `V`.

Adhering to Rel's fully relational design, `crossProduct` is a relation. However, *conceptually*, it is second-order, as its first two columns contain relations instead of values:

crossProduct					
{0}	{0}	0	0		
{0}	{1}	0	1		
{1}	{0}	1	0		
...					
{(1,2), (3,4)}	{(5,6)}	1	2	5	6
{(1,2), (3,4)}	{(5,6)}	3	4	5	6
...					

The relation `crossProduct` is also infinite. It has infinitely many rows, because it contains infinitely many relations in its first column alone. It also has infinitely many columns, because the relations in the first two columns do not have an upper bound on their arity. As such, the tuples in their cross product become arbitrarily long.

Remark. Readers familiar with the nested relational data model may also think of `crossProduct` as a nested relation. Such relations cannot be used as base relations nor query outputs, but conceptually they are necessary when relations themselves can be passed as parameters.

Relational Application. We can write `mother("John", "Emma")` to test if $\langle \text{"John"}, \text{"Emma"} \rangle \in \text{mother}$, which is the standard notation in our field for *relational atoms* [3, 24]. Another, more general way of understanding this is seeing `mother` as a Boolean function that, given two input parameters, a and b , returns whether $\langle a, b \rangle \in \text{mother}$. This principle works for all kinds of relations we have seen, including second-order ones. For example, if relation `R` contains the tuple $\langle 1, 2 \rangle$, and relation `S` contains the tuple $\langle 5, 6 \rangle$, then

```
crossProduct(R, S, 1, 2, 5, 6)
```

evaluates to true, because the second-order tuple $\langle R, S, 1, 2, 5, 6 \rangle$ is in the `crossProduct` relation. What we just described is one form of *relational application* in Rel. We call it *full application*, because the syntax with round brackets requires that all arguments be given.

Rel also supports *partial application*, which uses square brackets instead of round ones, and returns all suffixes in a relation that are consistent with a given prefix. For example, `parent["John"]` contains $\{\langle \text{"Emma"} \rangle, \langle \text{"Joe"} \rangle\}$ for the database in Figure 1, since both $\langle \text{"John"}, \text{"Emma"} \rangle$ and $\langle \text{"John"}, \text{"Joe"} \rangle$ are in `parent`. The same principle holds for second-order relations. So,

```
crossProduct[R, S]
```

evaluates to the cross product of the relations R and S .

In fact, since the cross product of two relations is so common in database applications, Rel has the syntactic sugar

```
(R,S)
```

to abbreviate `crossProduct[R, S]`. For example, `(mother, parent)` is the cross product of `mother` and `parent`, while `(“Joe”, 40)` is the singleton relation $\{\langle \text{"Joe"}, 40 \rangle\}$.

We need not provide all arguments in partial application, which thus evaluates to a relation. For example, `crossProduct[{1}]` is a second-order relation that can be applied further. To illustrate this, take the expression `{crossProduct[{1}]}[cat]` using the relation `cat` from Figure 1. This will return $\{\langle 1, \text{"Snuffles"} \rangle\}$. We can also name the expression by writing `def CP1 {crossProduct[{1}]}`, after which `CP1[cat]` returns the same result. This is a relational version of *currying* from functional programming. Indeed, any relation R can be seen as a function $a \mapsto R_a$ where $R_a = \{\langle \bar{b} \rangle \mid \langle a, \bar{b} \rangle \in R\}$. By passing a as a parameter to R , we obtain a new relation R_a that takes one parameter less.

If the relation resulting from partial application has arity zero, the result is either $\{\langle \rangle\}$ (the relation with the empty tuple) or $\{\}$ (the empty relation). In Rel, these two relations encode Boolean values true and false, respectively, as usual in the relational data model [3]. Therefore, full application computes the same result as partial application if all arguments are provided.

Expressions and Formulas. With partial application, we can write *RelExpressions* that evaluate to relations rather than Booleans. In fact, Rel calls *Formula* any *RelExpression* that evaluates to a Boolean, and this distinction allows for some syntactic sugar. For instance, if we want to condition the evaluation of a *RelExpression* on the truth of a *Formula*, we can write

```
RelExpression where Formula
```

and one may verify that it is equivalent to the cross product $(RelExpression, Formula)$. Indeed, taking the cross product of any relation R with $\{\langle \rangle\}$ is R itself, and taking the cross product of R with the empty relation is empty.

Abstraction. Until now we used Rel rules of the form:

```
def RName(VariableList): Formula (1)
```

Actually, Rel rules can also have the form:

```
def RName[VariableList]: RelExpression (2)
```

In both forms, the expression following `RName` is called *Abstraction* and evaluates to a relation. In general, *Abstraction* can have one of the following forms:

```
{ ( VariableList ) : Formula } (3a)
```

```
{ [ VariableList ] : RelExpression } (3b)
```

and the outer curly braces can be omitted to allow the forms (1) or (2).

Abstractions of the form (3a) are inspired by *set comprehension* from mathematics, where we write, e.g., $\{n \in \mathbb{N} : \exists m \in \mathbb{N} \text{ such that } n = 2m\}$ for the set of even numbers. For example,

```
{ (x,y) : father(x,"Joe") and ancestor("Joe",y) }
```

defines a binary relation linking children of “Joe” with their paternal ancestors.

Abstractions of the form (3b) use square brackets and allow an arbitrary *RelExpression* on the right. For example

```
{ [x,y] : height[y] - height[x] where parent(x,y) } (4)
```

defines a ternary relation containing, for each child-parent pair, the amount that the parent is taller than the child. Here we iterate over all possible pairs $\langle v_x, v_y \rangle$ of values for x and y and compute the expression on the right of the colon. That is, for each pair $\langle v_x, v_y \rangle \in \text{parent}$ we find the tuples $\langle v_x, h_x \rangle$ and $\langle v_y, h_y \rangle$ in `height` and add $\langle v_x, v_y, h_y - h_x \rangle$ to the result. On the database of Figure 1, we would add the tuple $\langle \text{"Kate"}, \text{"Joe"}, -3 \rangle$ to the result. Notice that in general the expression on the right of the colon does not need to compute a single value but can evaluate to any set of tuples.

We call this operation (*relational*) *abstraction* to draw a parallel with functional abstraction in functional programming languages. Indeed, lambda-abstraction $\lambda x.e$ denotes a function that for argument x computes $e(x)$. This function can be viewed as a relation, namely the set of pairs $\langle x, e(x) \rangle$. In relational abstraction, the difference is that e can be an expression that produces a relation instead of a function. The semantics of the abstraction then is the set of tuples $\langle x, y_1, \dots, y_k \rangle$ with $\langle y_1, \dots, y_k \rangle \in e(x)$.

Using the syntax given in (2), relational abstraction may be used to define rules. For instance, the next rule defines `parentHeightDiff` to be the relation previously computed by (4):

```
def parentHeightDiff[x,y] :  
  height[y] - height[x] where parent(x,y)
```

Let us stress that the rule above adds *triplets* to `parentHeightDiff`: the first two columns correspond to $[x,y]$, and the rest corresponds to the arity of the *RelExpression* right of the colon (which in this case is one more column).

5. AGGREGATION

Aggregation, a crucial component of query languages, is implemented in Rel as part of the standard library. For instance, `sum[R]` returns the sum of the elements in the last column of relation R . This finds justification in the underlying GNF for R : if we want to aggregate values in R , then R is going to be a set of key/value pairs, where the last column contains the values. The usual aggregates `sum`, `count`, `min`, `max`, `average` (and more) are available.

Aggregation with Grouping under Set Semantics. One often hears the argument that bag semantics is necessary for defining aggregation correctly. However, bags are only required if one looks solely at the aggregated column, and not at tuples in the relation that contain the aggregated column. We illustrate this using the schema from Section 2.

Assume that we wish to compute the average height of children of each person p . This is defined using relational application and abstraction:

```
def avgChildHeight[p] :
  average[(c,h) : parent(c,p) and height(c,h)]
```

The example illustrates how set semantics interacts with grouping: the argument of `average` evaluates to a set of pairs $\langle \text{child}, \text{height} \rangle$, which makes multiple same-height children of a person p all contribute to the average computed over the last column. It also illustrates how grouping by persons p is done using relational abstraction on $[p]$. Note that if we had only kept the heights in the argument of `average`, we would have computed something different:

```
def notTheAvgChildHeight[p] :
  average[(h) : exists((c) | parent(c,p) and height(c,h))]
```

Indeed, here, for each p , we define the set $\{h : h \text{ is a height of some child of } p\}$ and take the average of the values in the set. As such, this definition counts two children with the same height only once due to set semantics.

Therefore we can conclude that *aggregation works with set semantics*. With the right language design, bags are not needed to cleanly define aggregation.

Aggregation and Linear Algebra. The design of aggregation and partial application in Rel makes it very natural to write code for linear algebra. Let us model vectors as binary relations: the first column holds a position, and the second holds the value at that position. Matrices are encoded as ternary relations: the first two columns encode a position (row and column), and the third the value. Two such examples of encoding a vector $\mathbf{v}$ of length 4 and a 2×2 -matrix $\mathbf{M}$ are shown below.

$$\underbrace{\begin{pmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \end{pmatrix}}_V \Rightarrow \begin{array}{c|c} & V \\ \hline 1 & v_1 \\ 2 & v_2 \\ 3 & v_3 \\ 4 & v_4 \end{array} \quad \underbrace{\begin{pmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{pmatrix}}_M \Rightarrow \begin{array}{c|c|c} & & M \\ \hline 1 & 1 & m_{11} \\ 1 & 2 & m_{12} \\ 2 & 1 & m_{21} \\ 2 & 2 & m_{22} \end{array}$$

The same principle works for tensors: k -dimensional tensors are encoded as $(k+1)$ -ary relations, where the first k columns encode the tensor coordinates, and the last one the value.

Using `sum`, we can now define the sum of the elements of row i as:

```
def rowSum[{A},i] : {sum[[j] : A[i,j]]}
```

It is not a coincidence that the code for `rowSum[A,i]` looks similar to the mathematical expression $\sum_j a_{ij}$. The ranges of i and j are guarded by the first and second column of A , respectively. Hence, for each value of i that appears in the first column of relation A , the subexpression $[j] : A[i,j]$ evaluates to the binary relation $\{\langle j, a_{ij} \rangle \mid j \text{ appears in the second column of } A\}$. On the relation

$$\begin{array}{c|c|c} & & M \\ \hline 1 & 1 & 1 \\ 1 & 2 & 2 \\ 2 & 1 & 0 \\ 2 & 2 & 4 \end{array} \quad \left(\text{representing the matrix } \begin{pmatrix} 1 & 2 \\ 0 & 4 \end{pmatrix} \right)$$

`rowSum[M,1]` returns 3. We do not have to provide the row numbers: the expression `rowSum[M]` returns $\{\langle 1, 3 \rangle, \langle 2, 4 \rangle\}$.

Missing Values. It is interesting to observe that `rowSum` computes exactly the same results if the tuple $\langle 2, 1, 0 \rangle$ were missing from M . This means that representing sparse matrices is automatically supported, although one has to be careful not to exclude too many values (e.g., inferring the dimensions of a matrix that only consists of zero-entries is not possible if all tuples are missing). We discuss more complex examples in linear algebra, such as matrix multiplication and PageRank in Section 6.

6. CODE EXAMPLES

We now discuss a few slightly larger code examples that illustrate several prominent features of Rel programming. For the sake of compactness, this section's code will make use of *wildcard* ($_$), which in Rel, as in Datalog, denotes an anonymous existentially quantified variable, e.g.,

```
(y) : R(.,y,...)
```

is equivalent to $(y) : \text{exists}((x,z,...) \mid R(x,y,z,...))$ and evaluates to the second column of relation R .

6.1 Relational and Linear Algebra

Notation for linear algebra (LA) and relational algebra (RA) as well as languages such as APL [20] and FP [4] allows writing programs using a set of generally useful primitives and avoiding named variables, a style called *point-free* or *tacit programming*. The point-free style is also quite popular in business intelligence tools. Rel supports point-free style via libraries rather than language extensions, which we now demonstrate with libraries for RA and LA.

Relational Algebra. A simple example of relations defined in the library are familiar relational algebra operators. Cartesian product is defined as the `crossProduct` relation that we already discussed in Section 4. We also mentioned there that Rel uses the infix notation $\{A, B\}$ to denote the Cartesian product of A and B .

The union of two relations can also be defined in the library as follows:

```
def union({A},{B},x...) : A(x...) or B(x...)
```

Just like product, union has an infix shorthand: $\{A; B\}$. This way we can build arbitrary relations from constants; e.g.,

```
{(1,2,3) ; (4,5,6) ; (7,8,9)}
```

evaluates to $\{\langle 1, 2, 3 \rangle, \langle 4, 5, 6 \rangle, \langle 7, 8, 9 \rangle\}$. The remaining set operator of relational algebra — difference — is similarly defined by the following code:

```
def minus({A},{B},x...) : A(x...) and not B(x...)
```

A selection operator simply takes a relation and a condition — which would be an infinite relation — and returns their intersection:

```
def select({A},{Cond},x...) : A(x...) and Cond(x...)
```

Consider, for example, a relational algebra expression $\sigma_{1=2}(R \times S) \cup B$. To express this in Rel in point-free style, we first need an infinite set of tuples whose first and second components are equal:

```
def cond12(x1,x2,x...) : {x1=x2}
```

and then define the entire expression as follows:

```
union[select[crossProduct[R,S],cond12],B]
```

Projection can be easily expressed in Rel using abstraction. For instance, the projection of a relation on the first and third attribute may be expressed by the following code:

```
def proj13 ({A},x,y) : A(x,.,y,...)
```

Linear Algebra. Recall the relational representation of vectors, matrices, and tensors from Section 5. We now dive a bit deeper into linear algebra and look at slightly more complex examples. Given two vectors $\mathbf{u} = (u_1, \dots, u_n)$ and $\mathbf{v} = (v_1, \dots, v_n)$, their scalar product is $\mathbf{u} \cdot \mathbf{v} = \sum_k u_k v_k$. Rel's definition below mimics the mathematical definition:

```
def scalarProd[{U},{V}] : {sum[[k] : U[k]*V[k]]}
```

The range of k is guarded by the first columns of U and V , hence the subexpression $[k] : U[k]*V[k]$ evaluates to the

binary relation $\{(i, u_i v_i) \mid i \in \{1, \dots, n\}\}$. Recall that the `sum` aggregate computes the sum of the values in the last column of its argument (see Section 5). Hence this rule indeed computes $\sum_k u_i v_i = \mathbf{u} \cdot \mathbf{v}$.

The same approach makes it easy to define matrix multiplication. Recall that if $\mathbf{M} = \mathbf{A} \cdot \mathbf{B}$, then its entry m_{ij} in the i th row and j th column is $\sum_k a_{ik} \cdot b_{kj}$, as reflected in the following Rel definition:

```
def matProd[{A},{B},i,j] : {sum[[k] : A[i,k]*B[k,j]]}
```

Similarly, $\mathbf{A} \cdot \mathbf{v}$ for a matrix $\mathbf{A}$ and a vector $\mathbf{v}$ is defined in Rel by what follows:

```
def matVectProd[{A},{V},i] : {sum[[k] : A[i,k]*V[k]]}
```

An advantage of point-free code is that it is more robust under changes of the underlying data. For instance, `Union[R,S]` and `MatProd[A,B]`, work independently of the arities of the relations $\mathbf{R}$ and $\mathbf{S}$ or the dimensions of the matrices $\mathbf{A}$ and $\mathbf{B}$.

6.2 Graph Algorithms

All Pairs Shortest Paths. Next, we demonstrate code for the all pairs shortest path problem. The following code expects two relation variables $\mathbf{V}$ and $\mathbf{E}$, representing the finite set of vertices and edges (which are pairs of vertices) of a graph.

```
def APSP({V},{E},x,y,0) : V(x) and V(y) and x = y
def APSP({V},{E},x,y,i) :
  exists ((z in V) | E(x,z) and APSP(V,E,z,y,i-1)) and
  not exists ((j in Int) | j < i and APSP(V,E,x,y,j))
```

The first rule states that the shortest path from a node to itself has length 0. The second rule states that the shortest paths from x to y have length i if there exists an out-neighbor z of x such that the shortest paths from z to y have length $i - 1$ and we have not already discovered paths shorter than i from x to y . Although we use quantification over the entire relation `Int`, the Rel engine will not loop over all integers to compute the query.

We can also define `APSP` using aggregation and abstraction, as shown below.

```
def APSP({V},{E},x,y,0) : V(x) and V(y) and x = y
def APSP({V},{E},x,y) :
  min[(j) : exists((z) | E(x,z) and APSP(V,E,z,y,j-1))]
```

PageRank. Using the vector and matrix encoding from Section 5 and basic linear algebra functions from Section 6.1, we now write (a simplified version of) the PageRank algorithm in Rel. We also illustrate how a Rel program can perform a number of steps until a stopping condition is met.

```
def vector[d,i] : 1.0/d where range(1,d,1,i)
def dimension[{A}] : max[(k) : A(k,_,_) or A(_,k,_)]
def delta[{U},{V}] : max[[k] : absolute[U[k] - V[k]]]
```

```
def next[{A},{V}] : {matVectProd[A,V]}
def stop[{A},{V}] : {delta[next[A,V],V] < 0.005}

def pageRank[{A}] :
  {vector[dimension[A]] where empty(pageRank[A])}
def pageRank[{A}] :
  {next[A,pageRank[A]] where not empty(pageRank[A])
   and not stop(A,pageRank[A])}
def pageRank[{A}] : {pageRank[A] where
  not empty(pageRank[A]) and stop(A,pageRank[A])}
```

Here, `absolute` is defined as in Section 3, `empty` is the emptiness test given by

```
def empty({R}) : not exists((x...) | R(x...))
```

and the predicate `range(k,d,1,i)` is true for $i = k, k + 1, \dots, d$. Given an input graph, if $\mathbf{M}$ is the usual input matrix

of the PageRank algorithm whose entry (i, j) is $1/k$, where k is the j -th vertex's out-degree, then `pageRank[M]` iteratively computes PageRank of the graph, until the delta between two consecutive iterations is at most 0.005.

6.3 Querying Property Graphs

Property graphs, a popular graph data model, can be easily represented relationally. A possible GNF encoding is as follows: we have relations `node(u)` and `edge(e)` for the graph's nodes and edges, as well as `src(e,u)`, `tgt(e,v)` to indicate the source and target node of each edge e . Simplifying for the examples here, assume that nodes and edges have one property $\mathbf{V}$. This is encoded by a relation `V(x,w)` that pairs each node or edge x with a value w .

To start with, we define the reachability query that can be applied to an arbitrary binary relation $\mathbf{E}$:

```
def reachable({E},x,y) :
  E(x,y) or exists((z) | E(x,z) and reachable(E,z,y))
```

Assume that we want to find node pairs (u, v) of a property graph such that there is a path from u to v with increasing values on nodes. We start with a binary relation `goodNodePair` of nodes (u, v) connected by an edge so that the value in v is greater than the value in u :

```
def goodNodePair (u, v) : exists( (e in edge) |
  src(e,u) and tgt(e,v) and V[u] < V[v] )
```

The expression `reachable[goodNodePair]` gives us the desired node pairs. Assume now that we want to find node pairs (u, v) such that there is a path from u to v with increasing values on edges. This seemingly very similar query is however not naturally expressible with pattern matching facilities of existing graph query languages [15]. At the same time, it is very easy to define in Rel:

```
def goodEdgePair (e1,e2) : edge(e1) and edge(e2)
  and tgt[e1] = src[e2] and V[e1] < V[e2]
```

Now, the expression `reachable[goodEdgePair]` gives us edge pairs (e_1, e_2) such that there is a path from the source u of e_1 to the target v of e_2 with increasing values on edges. To get node pairs (u, v) themselves, we can use:

```
def incValOnEdges(u,v) :
  exists((e1,e2) | reachable(goodEdgePair,e1,e2)
  and src(e1,u) and tgt(e2,v))
```

6.4 Semantics and Business Logic

In the introduction we stated that Rel can express complete business logic by bringing semantics close to the data, and making use of various reasoners. Let us turn this slogan into something more tangible and consider the question: “How can we price our items to maximize income from sales in western states in the next quarter?” This is not a traditional database query for at least three different reasons. First, it refers to concepts that are not yet completely defined. For example, which states are Western? What quarter is next? Second, it needs data that we do not yet have in the database: sales for the next quarter. This will be obtained by a *predictive reasoner* that we can invoke. Third, how do we optimize? This will be done by a *prescriptive reasoner*, or solver. Finally, answering our question is a multi-step process involving database queries for defining semantics, as well as data that will be used by reasoners — all this data comes in shape of relations produced and consumed by Rel.

Assume (simplified for the paper) that we have base relations `itemPrice(item, state, quarter, price)`,

`sales(item, day, state, numItems)`, `inventory(item, day, state, numItems)` of historical data. Semantic concepts used in queries are relations defined in Rel, e.g.,

```
def dayQuarter(day, quarter) : ...
def stateRegion {...}
```

These are simple Rel queries the first of which maps a day to a quarter based on factors such as the beginning of the fiscal year, and the second contains pairs such as ("CA", "Western"), ("NY", "Eastern") etc.

So how to find the prices that maximize our income from sales? For this example, we assume a simplified model where the number of items i sold in a given quarter is $s_i = m_i - \sum_j e_{ij} p_j$, where m_i is the inventory of item i at the beginning of the quarter and p_i is its price, and e_{ij} is the *elasticity coefficient* giving how the price of item j affects sales of item i . Based on historical data, we will be predicting elasticity coefficients for the next quarter. We will then use them to find optimal prices.

First we need the data from past quarters to feed to our predictive reasoner. This is obtained by querying the database. For example, to obtain the relevant historical inventories m_i (in the Western states) at the beginning of each quarter, we write:

```
def invW[item, quarter] : sum[ [day, state] :
  inventory[item, day, state] where firstDay(day, quarter)
  and stateRegion[state]="Western"]
```

where `firstDay(d, q)` says that d is the first day of quarter q . Likewise, we define a relation `totalSalesW(item, amount)` of total sales s_i in the Western region (similarly obtained by aggregation). The historical values of p_j come from `itemPrice`. Notice that by creating semantic definitions for concepts like `stateRegion`, we can continue using exact same queries even when the semantics changes, say if "HI" is added to Western states, or the beginning of the fiscal year is modified.

Second, these relations are passed as relational parameters to a predictive reasoner that computes a matrix `e1` for the elasticity coefficients e_{ij} . The matrix is represented as a relation `e1[i, j, coeff]`, as in Section 5.

Third, we must now find prices p_i that maximize total sales income in the next quarter, which is $\sum_i s_i p_i = \sum_i (m_i p_i - \sum_j e_{ij} p_i p_j)$. We obtain the m_i 's (for the next quarter `next_q`) using the expression `{[i]:invW[i, next_q]}`, and pass it together with `e1` to a quadratic programming solver (a Rel library). It returns the optimal prices p_i in the form of a relation `result(item, optimalprice)`.

Thus, using appropriate libraries in Rel, one can program the entire application end to end in a purely relational fashion, with all the important concepts: semantic definitions that stay close to data, parameters for predictive and prescriptive reasoners, and their outputs, being relations defined in Rel. By consistently applying a design based on a small number of powerful primitives, Rel is designed to make this programming task easy for both human programmers and GenAI.

Acknowledgments

We are deeply indebted to Martin Bravenboer, who is one of the original designers of Rel and to the RelationalAI product and field engineering teams for being early users of Rel and providing invaluable feedback that helped improve it.

7. REFERENCES

- [1] M. Aref, P. Guagliardo, G. Kastrinis, L. Libkin, V. Marsault, W. Martens, M. McGrath, F. Murlak, N. Nystrom, L. Peterfreund, A. Rogers, C. Sirangelo, D. Vrgoč, D. Zhao, and A. Zreika. Rel: A programming language for relational data. In *SIGMOD*, pages 283–296. ACM, 2025.
- [2] M. Aref, B. ten Cate, T. J. Green, B. Kimelfeld, D. Olteanu, E. Pasalic, T. L. Veldhuizen, and G. Washburn. Design and implementation of the LogicBlox system. In *SIGMOD*, pages 1371–1382, 2015.
- [3] M. Arenas, P. Barceló, L. Libkin, W. Martens, and A. Pieris. *Database Theory*. Open source at <https://github.com/pdm-book/community>, 2022.
- [4] J. W. Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Commun. ACM*, 21(8):613–641, 1978.
- [5] R. T. Boute. Functional declarative language design and predicate calculus: a practical approach. *ACM Trans. Program. Lang. Syst.*, 27(5):988–1047, 2005.
- [6] S. Brass and C. Goldberg. Semantic errors in SQL queries: A quite complete list. *J. Syst. Softw.*, 79(5):630–644, 2006.
- [7] S. Cass. The top programming languages 2024. <https://spectrum.ieee.org/top-programming-languages-2024>.
- [8] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970.
- [9] E. F. Codd. A database sublanguage founded on the relational calculus. In *ACM-SIGFIDET Workshop on Data Description 1971*, pages 35–68. ACM, 1971.
- [10] E. F. Codd. Relational completeness of data base sublanguages. *IBM Research Report*, RJ987, 1972.
- [11] E. F. Codd. Extending the database relational model to capture more meaning. *ACM Trans. Database Syst.*, 4(4):397–434, 1979.
- [12] E. F. Codd. Relational database: A practical foundation for productivity. *Commun. ACM*, 25(2):109–117, 1982.
- [13] C. J. Date. *Database Technology: Nulls Considered Harmful*. Technics Publications, 2024.
- [14] C. J. Date, H. Darwen, and N. A. Lorentzos. *Temporal data and the relational model*. Elsevier, 2002.
- [15] A. Gheerbrant, L. Libkin, L. Peterfreund, and A. Rogova. GQL and SQL/PGQ: theoretical models and expressive power. *Proc. VLDB Endow.*, 18(6):1798–1810, 2025.
- [16] T. J. Green. Bag semantics. In *Encyclopedia of Database Systems, Second Edition*. 2018.
- [17] P. Guagliardo, L. Libkin, V. Marsault, W. Martens, F. Murlak, L. Peterfreund, and C. Sirangelo. Queries with external predicates. In *ICDT*, pages 22:1–22:20, 2025.
- [18] T. A. Halpin and T. Morgan. *Information modeling and relational databases*. Morgan Kaufmann, 2008.
- [19] Y. E. Ioannidis and R. Ramakrishnan. Containment of conjunctive queries: Beyond relations as sets. *ACM Trans. Database Syst.*, 20(3):288–324, 1995.
- [20] K. E. Iverson. *A Programming Language*. John Wiley & Sons, Inc., USA, 1962.
- [21] T. S. Jayram, P. G. Kolaitis, and E. Vee. The containment problem for REAL conjunctive queries with inequalities. In *PODS*, pages 80–89, 2006.
- [22] L. Libkin and L. Peterfreund. SQL nulls and two-valued logic. In *PODS*, pages 11–20, 2023.
- [23] J. Marcinkowski and P. Ostropolski-Nalewaja. Bag semantics query containment: The CQ vs. UCQ case and other stories. *PACMOD*, 3(5):1–24, 2025.
- [24] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill, 2003.
- [25] G. L. Steele. Growing a language. *High. Order Symb. Comput.*, 12(3):221–236, 1999.
- [26] E. Toussaint, P. Guagliardo, L. Libkin, and J. Sequeda. Troubles with nulls, views from the users. *Proc. VLDB Endow.*, 15(11):2613–2625, 2022.