SIGMOD Officers, Committees, and Awardees

Chair
Divyakant Agrawal
Department of Computer Science
UC Santa Barbara
Santa Barbara, California
USA
+1 805 893 4385
agrawal <at> cs.ucsb.edu

Vice-Chair
Fatma Ozcan
Systems Research Group
Google
Sunnyvale, California
USA
+1 669 264 9238
Fozcan <at> google.com

Secretary/Treasurer
Rachel Pottinger
Department of Computer Science
University of British Columbia
Vancouver
Canada
+1 604 822 0436
Rap <at> cs.ubc.ca

SIGMOD Executive Committee:

Divyakant Agrawal (Chair), Fatma Ozcan (Vice-chair), Rachel Pottinger (Treasurer), Juliana Freire (Previous SIGMOD Chair), Chris Jermaine (SIGMOD Conference Coordinator and ACM TODS Editor in Chief), Rada Chirkova (SIGMOD Record Editor), Angela Bonifati (2022 SIGMOD PC co-chair), Amr El Abbadi (2022 SIGMOD PC co-chair), Floris Geerts (Chair of PODS), Sihem Amer-Yahia (SIGMOD Diversity and Inclusion Coordinator), Sourav S Bhowmick (SIGMOD Ethics)

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:

Huiping Cao (SIGMOD Record), Georgia Koutrika (Blogging), Wim Martens (PODS)

SIGMOD Record Editor-in-Chief:

Rada Chirkova, NC State University

SIGMOD Record Associate Editors:

Lyublena Antova, Marcelo Arenas, Manos Athanassoulis, Renata Borovica-Gajic, Vanessa Braganholo, Susan Davidson, Aaron J. Elmore, Wook-Shin Han, Wim Martens, Kyriakos Mouratidis, Dan Olteanu, Tamer Özsu, Kenneth Ross, Pınar Tözün, Immanuel Trummer, Yannis Velegrakis, Marianne Winslett, and Jun Yang

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

SIGMOD Awards Committee:

H.V. Jagadish (Chair), Stefano Ceri, Yanlei Diao, Samuel Madden, Volker Markl, Sayan Ranu, Barna Saha, Wang-Chiew Tan

Jim Gray Doctoral Dissertation Award Committee:

Wolfgang Lehner (co-chair), Gustavo Alonso (co-chair), Azza Abouzied, Daniel Deutch, Evaggelia Pitoura, Xiaofang Zhou, Huanchen Zhang, and Chenggang Wu

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

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 Nandrekar-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)

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)
Jeffrey Ullman (1996)
Raghu Ramakrishnan (1999)
Daniel Rosenkrantz (2001)
Surajit Chaudhuri (2004)
Hans-Jörg Schek (2007)
David Lomet (2010)
H.V. Jagadish (2013)
Samuel Madden (2016)
Ahmed Elmagarmid (2019)
Stratos Idreos (2020)
Dennis Shasha (2020)
K. Selcuk Candan (2023)

Gio Wiederhold (1995)
Avi Silberschatz (1997)
Michael Carey (2000)
Richard Snodgrass (2002)
Hongjun Lu (2005)
Klaus R. Dittrich (2008)
Gerhard Weikum (2011)
Kyu-Young Whang (2014)
Yannis E. Ioannidis (2017)
Philipe Bonnet (2020)
Stefan Manegold (2020)
Divesh Srivastava (2021)

Yahiko Kambayashi (1995)
Won Kim (1998)
Laura Haas (2000)
Michael Ley (2003)
Tamer Özsu (2006)
Beng Chin Ooi (2009)
Marianne Winslett (2012)
Curtis Dyreson (2015)
Z. Meral Özsoyoğlu (2018)
Juliana Freire (2020)
Ioana Manolescu (2020)
Christian S. Jensen (2022)

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** *Winne*r: Peter Bailis. *Honorable Mention*: Immanuel Trummer
- **2018** *Winne*r: Viktor Leis. *Honorable Mention*: Luis Galárraga and Yongjoo Park
- **2019** *Winne*r: 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

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

[Last updated: June 1, 2023]

Guest Editors' Notes

Welcome to the March 2024 issue of the ACM SIGMOD Record, a special issue presenting papers and technical perspectives covering a selection of the best conference papers from 2023 in data management. These papers have been given the **2023 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 Angela Bonifati, Rada Chirkova, Alfons Kemper, Samuel Madden, Kenneth Ross (chair) and Yufei Tao. We solicited articles from PODS 2023, SIGMOD 2023, VLDB 2023, ICDE 2023, EDBT 2023, and ICDT 2023, 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 2023 Research Highlight Award winners. One of the articles was previously highlighted in the December 2023 issue of ACM SIGMOD Record, and so it does not appear here in the March 2024 issue.

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 2024 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.

The 2023 research highlight award winners include:

1) a method to compute histograms and heavy hitters over data streams in a differentially private setting while adding less noise than previous methods ("Better Differentially Private Approximate Histograms and Heavy Hitters using the Misra-Gries Sketch");
2) an algorithm to compute "robust" allocations of isolation levels to transactions that guarantees the safety of interleaved executions ("Allocating Isolation Levels to Transactions in a Multiversion Setting");
3) an analysis of methods for choosing between different conjunctive queries that fit a set of data examples ("Fitting Algorithms for Conjunctive Queries"); [See the December 2023 issue of SIGMOD Record]
4) a way to formulate worst-case optimal joins and conventional binary joins as special cases of a general join operator ("From Binary Join to Free Join");
5) a framework that manages samples for approximate query processing so that samples can be reused across queries even in the face of changing workloads ("Efficient and Reusable Lazy Sampling");

6) a novel machine learning model supporting data matching tasks for data integration ("Unicorn: A Unified Multi-tasking Matching Model");
7) a graph theoretic approach to modeling complex data flows in which privacy constraints require that certain nodes must be disconnected ("Graph Theory for Consent Management: A New Approach for Complex Data Flows");
8) a novel way to evaluate differential privacy mechanisms by taking real-world empirical data studies and asking whether the conclusions of those studies would have been visible using synthetic differentially-private data ("Epistemic Parity: Reproducibility as an Evaluation Metric for Differential Privacy");
9) a system that interactively and automatically transforms tables from spreadsheets and web pages into standard relational tables that can be processed by SQL ("Auto-Tables: Relationalize Tables without Using Examples");
10) a unified way of modeling incremental computation on data streams that enables incremental view maintenance ("DBSP: Incremental Computation on Streams and its Applications to Databases");

On behalf of the SIGMOD Record Editorial Board, we hope that you enjoy reading the March 2024 issue of the SIGMOD Record!

Angela Bonifati Rada Chirkova Alfons Kemper
Samuel Madden Kenneth Ross Yufei Tao

March 2024

Technical Perspective on 'Better Differentially Private Approximate Histograms and Heavy Hitters using the Misra-Gries Sketch'

Graham Cormode
University of Warwick, UK
G.Cormode@warwick.ac.uk

The topics of private data analysis and streaming data management have both been separately the focus of much study within the data management community for many years. However, more recently there have been studies which bring these two previously isolated topics together.

Although data streams and privacy might not seem to have much in common, it turns out that they share a symbiotic relationship. Within data streams, a common pattern is to design a compact data structure that summarizes the input that has been seen so far, and which can be updated quickly in order to reflect a small change. Within privacy, it is often helpful to build an intermediate representation of a data set, so that a small change to the input does not change the representation much. Moreover, both areas make use of the tools of randomized algorithms and probability to prove that their mechanisms give strong guarantees. Thus, data stream summaries can sometimes inspire private algorithms, and vice-versa.

A few prior examples of this phenomenon include analyzing the (differential) privacy of randomized projections via the Johnson-Lindenstrauss lemma [3] and the popular Flajolet-Martin count-distinct summaries [4]. This allows a variety of downstream data analysis to be performed using these data summaries, which are both compact and private. A major application of private data analysis is to find popular items among a large collection of observations, and tools for this deployed in major operating systems have made use of stream summaries (Bloom filters and Count-min sketches) [1, 2].

This work, by Lebeda and Tětek, first published in PODS 2023, also considers frequencies, and asks whether a popular streaming data structure can be made private. Specifically, it considers the Misra-Gries (MG) sketch, a simple and effective data stream summary that allows the frequency of each item to be estimated up to a $1/k$ additive error while using space proportional to k.

One perspective on the MG sketch is that it allows us to extract a sparse vector representation of the input. If we think of the input as defining a stochastic frequency vector x (i.e., $\|x\|_1 = 1$), then the MG summary provides a vector y such that the maximum error in any coordinate is bounded ($\|x - y\|_\infty \leq 1/k$) and it is sparse ($\|y\|_0 \leq k$). This is tantalizing, since given x in full there is an easy differentially private way to obtain a y that is sparse: add noise to each entry of x, and retain only those noisy entries that exceed a threshold. However, under the constraint of small space and streaming updates, it is not at all straightforward to obtain such a representation.

The approach here is very effective: essentially, with a simple post-processing step, the MG summary can be made (ϵ, δ)-differentially private. The algorithm is almost identical to the naive idealized algorithm: add noise to each entry of the sparse MG vector, and retain only those noisy entries that exceed a (noisy) threshold. The heavy lifting comes in the proof, which involves an intricate and clever case analysis to argue that the MG summary of two close inputs can only vary in a bounded way. Then, by relating the noise to this bound on deviation, differential privacy can be shown.

The implications of this result are powerful: we have a deeper understanding of this classical streaming algorithm, and a space-efficient technique to capture private frequency statistics of streams of data. Applications which process large volumes of data can form private representations based on the efficiently computable MG sketch.

The paper opens the door to further studies of questions at the intersection of streaming and privacy. What other practical and widely-used summaries can be made private by similar carefully calibrated noise addition? More broadly, what other models of computation can be made privacy-aware—in particular, computations that handle large volumes of data in support of "big data" and AI/ML workflows, and associated monitoring tasks? The nascent areas of federated analytics and federated learning in particular, which seek to support private evaluation of complex queries and modeling over distributed data, are ripe for making use of compact data summaries to minimize communication costs.

1. REFERENCES

[1] Apple Differential Privacy Team. Learning with privacy at scale. *Apple ML Research*, 2017.
[2] Ú. Erlingsson, V. Pihur, and A. Korolova. RAPPOR: randomized aggregatable privacy-preserving ordinal response. In *CCS*, 2014.
[3] K. Kenthapadi, A. Korolova, I. Mironov, and N. Mishra. Privacy via the Johnson-Lindenstrauss transform. *J. Priv. Confidentiality*, 5(1), 2013.
[4] A. D. Smith, S. Song, and A. Thakurta. The Flajolet-Martin sketch itself preserves differential privacy: Private counting with minimal space. In *Neural Information Processing Systems*, 2020.

Better Differentially Private Approximate Histograms and Heavy Hitters using the Misra-Gries Sketch

Christian Janos Lebeda
Basic Algorithms Research Copenhagen
IT University of Copenhagen
Copenhagen, Denmark
christian.j.lebeda@gmail.com

Jakub Tětek
Basic Algorithms Research Copenhagen
University of Copenhagen
Copenhagen, Denmark
j.tetek@gmail.com

ABSTRACT

We consider the problem of computing differentially private approximate histograms and heavy hitters in a stream of elements. In the non-private setting, this is often done using the sketch of Misra and Gries [Science of Computer Programming, 1982]. Chan, Li, Shi, and Xu [PETS 2012] describe a differentially private version of the Misra-Gries sketch, but the amount of noise it adds can be large and scales linearly with the size of the sketch; the more accurate the sketch is, the more noise this approach has to add. We present a better mechanism for releasing a Misra-Gries sketch under (ε, δ)-differential privacy. It adds noise with magnitude independent of the size of the sketch; in fact, the maximum error coming from the noise is the same as the best known in the private non-streaming setting, up to a constant factor. Our mechanism is simple and likely to be practical. In the full version of the paper we also give a simple post-processing step of the Misra-Gries sketch that does not increase the worst-case error guarantee. It is sufficient to add noise to this new sketch with less than twice the magnitude of the non-streaming setting. This improves on the previous result for ε-differential privacy where the noise scales linearly to the size of the sketch.

1. INTRODUCTION

Computing the histogram of a dataset is one of the most fundamental tasks in data analysis. At the same time, releasing a histogram may present significant privacy issues. This makes the efficient computation of histograms under privacy constraints a fundamental algorithmic question. Notably, differential privacy has become in recent years the golden standard for privacy, giving formal mathematical privacy guarantees. It would thus be desirable to have an efficient way of (approximately) computing histograms under differential privacy.

Histograms have been investigated thoroughly in the differentially private setting [1, 2, 7, 8, 10, 12, 14]. These algorithms start by computing the histogram exactly and they then add noise to ensure privacy. However, in practice, the amount of data is often so large that computing the histogram exactly would be impractical. This is, for example, the case when computing the histogram of high-volume streams such as when monitoring computer networks, online users, financial markets, and similar. In that case, we need an efficient streaming algorithm. Since the streaming algorithm would only compute the histogram approximately, the above-mentioned approach that first computes the exact histogram is infeasible. In practice, non-private approximate histograms are often computed using the Misra-Gries (MG) sketch [15]. The MG sketch of size k returns at most k items and their approximate frequencies $\hat{f}$ such that $\hat{f}(x) \in [f(x) - n/(k+1), f(x)]$ for all elements x where $f(x)$ is the true frequency and n is the length of the stream. This error is known to be optimal [5]. In this work, we develop a way of releasing a MG sketch in a differentially private way while adding only a small amount of noise.This allows us to efficiently and accurately compute approximate histograms in the streaming setting while not violating users' privacy. This can then be used to solve the heavy hitters problem in a differentially private way. Our result improves upon the work of Chan, Li, Shi, and Xu [6] who also show a way of privately releasing the MG sketch, but who need a greater amount of noise; we discuss this below.

In general, the issue with making approximation algorithms differentially private is that although we may be approximating a function with low global sensitivity, the algorithm itself (or rather the function it implements) may have a much larger global sensitivity. This increases the amount of noise required to achieve privacy using standard techniques. We get around this issue by exploiting the structure of the difference between the MG sketches for neighboring inputs. This allows us to prove that the following simple mechanism ensures (ε, δ)-differential privacy: (1) We compute the Misra-Gries sketch, (2) we add to each counter independently noise distributed as Laplace$(1/\varepsilon)$, (3) we add to all counters the same value, also distributed as Laplace$(1/\varepsilon)$, (4) we remove all counters smaller than $1 + 2\ln(3/\delta)/\varepsilon$. Specifically, we show that this algorithm satisfies the following guarantees:

THEOREM 11 (SIMPLIFIED). *The above algorithm is (ε, δ)-differentially private, uses $2k$ words of space, and returns a frequency oracle $\hat{f}$ with maximum error of $n/(k+1) + O(\log(1/\delta)/\varepsilon)$ with high probability for δ being sufficiently small.*

A construction for a differentially private Misra-Gries sketch has been given before by Chan et al. [6]. However, the more accurate they want their sketch to be (and the bigger it is), their approach has to add *more* noise. The reason is that they directly rely on the global ℓ_1-sensitivity of the sketch. Specifically, if the sketch has size k (and thus error $n/(k+1)$ on a stream of n elements), its global sensitivity is k, and they thus have to add noise of magnitude k/ε. Their mechanism ends up with an error of $O\left(k\log(d)/\varepsilon\right)$ for ε-differential privacy with d being the universe size. This can be easily improved to $O\left(k\log(1/\delta)/\varepsilon\right)$ for (ε,δ)-differential privacy with a thresholding technique similar to what we do in step (4) of our algorithm above. This also means that they cannot get more accurate than error $\Theta\left(\sqrt{n\log(1/\delta)/\varepsilon}\right)$, no matter what value of k one chooses. We achieve that the biggest error, as compared to the values from the MG sketch, among all elements is $O(\log(1/\delta)/\varepsilon)$ assuming δ is sufficiently small (we show more detailed bounds including the mean squared errors in Theorem 11). This is the same as the best private solution that starts with an exact histogram [14]. In fact, for any mechanism that outputs at most k heavy hitters there exists input with error at least $n/(k+1)$ in the streaming setting [5] and input with error at least $O(\log(\min(d,1/\delta))/\varepsilon)$ [2] under differential privacy.

In the full version of this paper, we also show how to achieve pure ε-differential privacy with error $n/(k+1)+O(\log(d)/\varepsilon)$. Therefore the error of our mechanisms is asymptotically optimal for approximate and pure differential privacy, respectively.

Chan et al. [6] use their differentially private Misra-Gries sketch as a subroutine for continual observation and combine sketches with an untrusted aggregator. Those settings are not a focus of our work but our algorithm can replace theirs as the subroutine, leading to better results also for those settings. However, the error from noise still increases linearly in the number of merges when the aggregator is untrusted. As a side note, we show that in the case of a trusted aggregator, the approach of [6] can handle merge operations without increasing error. While that approach adds significantly more noise than ours if we do not merge, it can with this improvement perform better when the number of merges is very large (at least proportional to the sketch size).

Another approach that can be used is to use a randomized frequency oracle to recover heavy hitters. However, it seems hard to do this with the optimal error size. In its most basic form [11, Appendix D], this approach needs noise of magnitude $\Theta(\log(d)/\varepsilon)$, even if we have a sketch with sensitivity 1 (the approach increases the sensitivity to $\log(d)$, necessitating the higher noise magnitude), leading to maximum error at least $\Omega(\log(k)\log(d)/\varepsilon)$. Bassily, Nissim, Stemmer, and Guha Thakurta [3] show a more involved approach which reduces the maximum error coming from the noise to $\Theta((\log(k)+\log(d))/\varepsilon)$, but at the cost of increasing the error coming from the sketch by a factor of $\log(d)$. This means that even if we had a sketch with error $\Theta(n/k)$ and sensitivity 1, neither of these two approaches would lead to optimal guarantees, unlike the algorithm we give in this paper.

See https://github.com/JakubTetek/Differentially-Private-Misra-Gries for sample implementations of the algorithms we present in this paper.

2. TECHNICAL OVERVIEW

Misra-Gries sketch. Since our approach depends on the properties of the MG sketch, we describe it here. Readers familiar with the MG sketch may wish to skip this paragraph. We describe the standard version; in Section 5 we use a slight modification, but we do not need that here.

Suppose we receive a sequence of elements from some universe. At any time, we will be storing at most k of these elements. Each stored item has an associated counter, other elements have implicitly their counter equal to 0. When we process an element, we do one of the following three updates: (1) if the element is being stored, increment its counter by 1, (2) if it is not being stored and the number of stored items is $< k$, store the element and set its counter to 1, (3) otherwise decrement all k counters by 1 and remove those that reach 0. The exact guarantees on the output will not be important now, and we will discuss them in Section 5.

Our contributions. We now sketch how to release an MG sketch in a differentially private way.

Consider two neighboring data streams $S=(S_1,\cdots,S_n)$ and $S'=(S_1,\cdots,S_{i-1},S_{i+1},\cdots,S_n)$ for some $i\in[n]$. At step $i-1$, the state of the MG sketch on both inputs is exactly the same. MG_S then receives the item S_i while $MG_{S'}$ does not. This either increments one of the counters of MG_S (possibly by adding an element and raising its counter from 0 to 1) or decrements all its counters. In ℓ_1 distance, the vector of the counters thus changes by at most k. One can show by induction that this will stay this way: at any point in time, $\|MG_S - MG_{S'}\|_1 \le k$. By a standard global sensitivity argument, one can achieve pure DP by adding noise of magnitude k/ε to each count. This is the approach used in [6]. Similarly, we could achieve (ε,δ)-DP by using the Gaussian mechanism [9] with noise magnitude proportional to the ℓ_2-sensitivity, which is $\sup_{S,S'}\|MG_S - MG_{S'}\|_2 \le \sqrt{k}$. We want to instead achieve noise with magnitude $O(1/\varepsilon)$ at each count. To this end, we need to exploit the structure of $MG_S - MG_{S'}$.

What we just described requires that we add the noise to the counts of all items in the universe, also to those that are not stored in the sketch. This results in the maximum error of all frequencies depending on the universe's size, which we do not want. However, it is known that this can be easily solved under (ε,δ)-differential privacy by only adding noise to the stored items and then removing values smaller than an appropriately chosen threshold [14]. This may introduce additional error – for this reason, we end up with error $O(\log(1/\delta)/\varepsilon)$. As this is a somewhat standard technique, we ignore this in this section, we assume that the sketches MG_S and $MG_{S'}$ store the same set of elements; the thresholding allows us to remove this assumption, while allowing us to add noise only to the stored items, at the cost of only getting approximate DP.

We now focus on the structure of $MG_S - MG_{S'}$. After we add to MG_S the element S_i, it either holds (1) that $MG_S - MG_{S'}$ is a vector of all 0's and one 1 or (2) that $MG_S - MG_{S'} = -\mathbf{1}^k$ (We use $\mathbf{1}^k$ the denote the dimension k vector of all ones). We show by induction that this will remain the case as more updates are done to the sketches (note that the remainders of the streams are the same). We do not sketch the proof here, as it is quite technical.

How do we use the structure of $MG_S - MG_{S'}$ to our advantage? We add noise twice. First, we independently add to each counter noise distributed as Laplace$(1/\varepsilon)$. Second, we add to all counters the same value, also distributed as Laplace$(1/\varepsilon)$. That is, we release $MG_S + \text{Laplace}(1/\varepsilon)^{\otimes k} + \text{Laplace}(1/\varepsilon)\mathbf{1}^k$ (For D being a distribution, we use $D^{\otimes k}$ to denote the k-dimensional distribution consisting of k independent copies of D). Intuitively speaking, the first noise hides the difference between S and S' in case (1) and the second noise hides the difference in case (2). We now sketch why this is so for worse constants: $2/\varepsilon$ in place of $1/\varepsilon$. When proving this formally, we use a more technical proof which leads to the better constant.

We now sketch why this is differentially private. Let m_S be the mean of the counters in MG_S for S being an input stream. We may represent MG_S as $(MG_S - m_S\mathbf{1}, m_S)$ (note that there is a bijection between this representation and the original sketch). We now argue that the ℓ_1-sensitivity of this representation is < 2 (treating the representation as a $(k+1)$-dimensional vector for the sake of computing the ℓ_1 distances). Consider the first case. In that case, the averages $m_S, m_{S'}$ differ by $1/k$. As such, $MG_S - m_S\mathbf{1}^k$ and $MG_{S'} - m_{S'}\mathbf{1}^k$ differ by $1/k$ at $k-1$ coordinates and by $1 - 1/k$ at one coordinate. The overall ℓ_1 change of the representation is thus

$$(k-1)\cdot\frac{1}{k} + (1-1/k) + 1/k = 2 - 1/k < 2.$$

Consider now the second case when $MG_S - MG_{S'} = -\mathbf{1}^k$. Thus, $MG_S - m_S = MG'_S - m_{S'}$. At the same time $|m_S - m_{S'}| = 1$. This means that the ℓ_1 distance between the representations is 1. Overall, the ℓ_1-sensitivity of this representation is < 2.

This means that adding noise from Laplace$(2/\varepsilon)^{\otimes k+1}$ to this representation of MG_S will result in ε-differential privacy. The resulting value after adding the noise is $(MG_S - m_S\mathbf{1}^k + \text{Laplace}(2/\varepsilon)^{\otimes k}, m_S + \text{Laplace}(2/\varepsilon))$. In the original vector representation of MG_S, this corresponds to $MG_S + \text{Laplace}(2/\varepsilon)^{\otimes k} + \text{Laplace}(2/\varepsilon)\mathbf{1}^k$ and, by post-processing, releasing this value is also differentially private. But this is exactly the value we wanted to show is differentially private!

3. PRELIMINARIES

Setup of this paper. We use $\mathcal{U}$ to denote a universe of elements. We assume that $\mathcal{U}$ is a totally ordered set of size d. That is, $\mathcal{U} = [d]$ where $[d] = \{1, \ldots, d\}$. Given a stream $S \in \mathcal{U}^{\mathbb{N}}$ we want to estimate the frequency in S of each element of $\mathcal{U}$. Our algorithm outputs a set $T \subseteq \mathcal{U}$ of keys and a frequency estimate c_i for all $i \in T$. The value c_j is implicitly 0 for any $j \notin T$. Let $f(x)$ denote the true frequency of x in the stream S. Our goal is to minimize the largest error between c_x and $f(x)$ among all $x \in \mathcal{U}$.

Differential privacy. Differential privacy is a rigorous definition for describing the privacy loss of a randomized mechanism introduced by Dwork, McSherry, Nissim, and Smith [8]. Intuitively, differential privacy protects privacy by restricting how much the output distribution can change when replacing the input from one individual. This is captured by the definition of neighboring datasets. We use the add-remove neighborhood definition for differential privacy.

Definition 1 (Neighboring Streams)**.** *Let S be a stream of length n. Streams S and S' are neighboring denoted $S \sim S'$ if there exists an i such that $S = (S'_1, \ldots, S'_{i-1}, S'_{i+1}, S'_{i+1}, \ldots, S'_{n+1})$ or $S' = (S_1, \ldots, S_{i-1}, S_{i+1}, \ldots, S_n)$.*

Definition 2 (Differential Privacy [9])**.** *A randomized mechanism $\mathcal{M} : \mathcal{U}^{\mathbb{N}} \to \mathcal{R}$ satisfies (ε, δ)-differential privacy if and only if for all pairs of neighboring streams $S \sim S'$ and all measurable sets of outputs $Z \subseteq \mathcal{R}$ it holds that*

$$\Pr[\mathcal{M}(S) \in Z] \leq e^{\varepsilon}\Pr[\mathcal{M}(S') \in Z] + \delta \ .$$

The privacy guarantees are parameterized by the values ε and δ. Smaller values for these privacy parameters imply stronger privacy guarantees. Differential privacy gives us a trade-off between privacy and utility. If we require strong privacy guarantees we can lower the privacy parameters. However, doing so forces us to add more noise to the output.

Several other properties such as composition, group privacy, and closure under post-processing follow directly from the Definition 2. We refer to the book by Dwork and Roth [9] for a longer introduction to these concepts.

Samples from a Laplace distribution are used in many differentially private algorithms, most notably the Laplace mechanism [8]. We write Laplace(b) to denote a random variable with a Laplace distribution with scale b centered around 0. We sometimes abuse notation and write Laplace(b) to denote the value of a random variable drawn from the distribution. Our mechanism also works with other noise distributions. We briefly discuss this in Section 5.2.

Definition 3 (Laplace distribution)**.** *The probability density and cumulative distribution functions of the Laplace distribution centered around 0 with scale parameter b are $f_b(x) = \frac{1}{2b}e^{-|x|/b}$, and $\Pr[\text{Laplace}(b) \leq x] = \frac{1}{2}e^{x/b}$ if $x < 0$ and $1 - \frac{1}{2}e^{-x/b}$ for $x \geq 0$.*

4. RELATED WORK

Chan et al. [6] show that the global ℓ_1-sensitivity of a Misra-Gries sketch is $\Delta_1 = k$. (They actually show that the sensitivity is $k+1$ but they use a different definition of neighboring datasets that assumes n is known. Applying their techniques under our definition yields sensitivity k.) They achieve privacy by adding noise with scale k/ε to all elements in the universe and keep the top-k noisy counts. This gives an expected maximum error of $O(k\log(d)/\varepsilon)$ with ε-DP for d being the universe size. They use the algorithm as a subroutine for continual observation and merge sketches with an untrusted aggregator. Those settings are not a focus of our work but our algorithm can replace theirs as the subroutine.

Böhler and Kerschbaum [4] worked on differentially private heavy hitters with no trusted server by using secure multi-party computation. One of their algorithms adds noise to the counters of a Misra-Gries sketch. They avoid adding noise to all elements in the universe by removing noisy counts below a threshold which adds an error of $O(\log(1/\delta)/\varepsilon)$. This is a useful technique for hiding differences in keys between neighboring sketches that removes the dependency on d in the error. Unfortunately, as we explain in the full version of our paper, their mechanism uses the wrong sensitivity. The sensitivity of the sketch is k. If the magnitude of noise and the threshold are increased accordingly the error of their approach is $O(k\log(k/\delta)/\varepsilon)$.

A closely related problem is that of implementing frequency oracles in the streaming setting under differential privacy. This has been studied in e.g. [11,16,17]. These approaches do not directly return the heavy hitters. The simplest approach for finding the heavy hitters is to iterate over the universe which might be infeasible. However, there are constructions for finding heavy hitters with frequency oracles more efficiently (see Bassily et al. [3]). However, as we discussed in the introduction, the approach of [3] leads to worse maximum error than what we get unless the sketch size is very large and the universe size is small.

5. DIFFERENTIALLY PRIVATE MISRA-GRIES SKETCH

In this section, we present our algorithm for privately releasing Misra-Gries sketches. We first present our variant of the non-private Misra-Gries sketch in Algorithm 1 and later show how we add noise to achieve (ε, δ)-differential privacy. The algorithm we use differs slightly from most implementations of MG in that we do not remove elements that have weight 0 until we need to re-use the counter. This will allow us to achieve privacy with slightly lower error.

At all times, k counters are stored as key-value pairs. We initialize the sketch with dummy keys that are not part of $\mathcal{U}$. This guarantees that we never output any elements that are not part of the stream, assuming we remove the dummy counters as post-processing.

The algorithm processes the elements of the stream one at a time. At each step one of three updates is performed: (1) If the next element of the stream is already stored the counter is incremented by 1. (2) If there is no counter for the element and all k counters have a value of at least 1 they are all decremented by 1. (3) Otherwise, one of the elements with a count of zero is replaced by the new element.

In case (3) we always remove the smallest element with a count of zero. This allows us to limit the number of keys that differ between sketches for neighboring streams as shown in Lemma 5. The choice of removing the minimum element is arbitrary but the order of removal must be independent of the stream so that it is consistent between neighboring datasets. The limit on differing keys allows us to obtain a slightly lower error for our private mechanism. However, it is still possible to apply our mechanism with standard implementations of MG. We discuss this in Section 5.1.

Algorithm 1: Misra-Gries (MG)

Input: Positive integer k and stream $S \in \mathcal{U}^{\mathbb{N}}$

$T \leftarrow \{d+1, \ldots, d+k\}$ // Start with k dummy counters
$c_i \leftarrow 0$ for all $i \in T$
foreach $x \in S$ **do**
  if $x \in T$ **then** // Branch 1
    $c_x \leftarrow c_x + 1$
  else if $c_i \geq 1$ *for all* $i \in T$ **then** // Branch 2
    $c_i \leftarrow c_i - 1$ for all $i \in T$
  else // Branch 3
    Let $y \in T$ be the smallest key satisfying $c_y = 0$
    $T \leftarrow (T \cup \{x\}) \setminus \{y\}$
    $c_x \leftarrow 1$
return T, c

The same guarantees about correctness hold for our version of the MG sketch, as for the original version. This can be easily shown, as the original version only differs in that it immediately removes any key whose counter is zero. Since the counters for items not in the sketch are implicitly zero, one can see by induction that the estimated frequencies by our version are exactly the same as those in the original version. We still need this modified version, as the set of keys it stores is different from the original version, which we use below. The fact that the returned estimates are the same however allows us to use the following fact

Fact 4 (Bose et al. [5])**.** *Let $\hat{f}(x)$ be the frequency estimates given by an MG sketch of size k for n being the input size. Then for all $x \in \mathcal{U}$, it holds $\hat{f}(x) \in [f(x) - n/(k+1), f(x)]$, where $f(x)$ is the true frequency of x.*

Note that this is optimal for any mechanism that returns a set of at most k elements. This is easy to see for an input stream that contains $k+1$ distinct elements each with frequency $n/(k+1)$ since at least one element must be removed.

We now analyze the value of $MG_S - MG_{S'}$ for S, S' being neighboring inputs (recall Definition 1). We will then use this in order to prove privacy. As mentioned in Section 4, Chan et al. [6] showed that the ℓ_1-sensitivy for Misra-Gries sketches is k. They show that this holds after processing the element that differs for neighboring streams and use induction to show that it holds for the remaining stream. Our analysis follows a similar structure. We expand on their result by showing that the sets of stored elements for neighboring inputs differ by at most two elements when using our variant of Misra-Gries. We then show how all this can be used to get differential privacy with only a small amount of noise.

Lemma 5. *Let $T, c \leftarrow \mathrm{MG}(k, S)$ and $T', c' \leftarrow \mathrm{MG}(k, S')$ be the outputs of Algorithm 1 on a pair of neigboring streams $S \sim S'$ such that S' is obtained by removing an element from S. Then $|T \cap T'| \geq k-2$ and all counters not in the intersection have a value of at most 1. Moreover, it holds that either (1) $c_i = c'_i - 1$ for all $i \in T'$ and $c_j = 0$ for all $j \notin T'$ or (2) there exists an $i \in T$ such that $c_i = c'_i + 1$ and $c_j = c'_j$ for all $j \neq i$.*

Proof. We sketch here the proof of the lemma. The full proof is in the full version of the paper.

Let $x = S_i$ be the element in stream S which is not in stream S'. Since the streams are identical in the first $i-1$ steps the sketches are clearly the same before step i. If there is no counter for x in the sketch and all k counters have a non-zero value we execute Branch 2 of Algorithm 1. The difference between the MG sketches for S and S' then corresponds to case (1) of the lemma. If there is a counter for x we execute Branch 1 of Algorithm 1. Finally, if there is no counter for x and at least one counter with weight 0 we execute Branch 3 of Algorithm 1. After executing either Branch 1 or 3 the difference between the MG sketches corresponds to case (2) of the lemma. In the full proof we restrict the difference between the two sketches to one of six states that all fall under either case (1) or (2). We show that if we are in one of the states at step j we will be in one of the states at step $j+1$ using a comprehensive case-by-case analysis. Since the difference between the sketches is in one of the states after step i the lemma holds by induction. □

Next, we consider how to add noise to release the Misra-Gries sketch under differential privacy. Recall that Chan et al. [6] achieves privacy by adding noise to each counter which scales with k. We avoid this by utilizing the structure of sketches for neighboring streams shown in Lemma 5. We sample noise from Laplace$(1/\varepsilon)$ independently for each counter, but we also sample one more random variable from the same distribution which is added to all counters. Small values are then discarded using a threshold to hide differences in the sets of stored keys between neighboring inputs. This is similar to the technique used by e.g. [14]. The algorithm takes the output from MG as input. We sometimes write $\text{PMG}(k, S)$ as a shorthand for $\text{PMG}(\text{MG}(k, S))$.

Algorithm 2: Private Misra-Gries (PMG)

Parameters: $\varepsilon, \delta > 0$
Input : Output from Algorithm 1: $T, c \leftarrow \text{MG}(k, S)$

$\tilde{T} \leftarrow \emptyset$
Sample $\eta \sim \text{Laplace}(1/\varepsilon)$
foreach $x \in T$ **do**
  $c_x \leftarrow c_x + \eta + \text{Laplace}(1/\varepsilon)$
  if $c_x \geq 1 + 2\ln(3/\delta)/\varepsilon$ **then**
    $\tilde{T} \leftarrow \tilde{T} \cup \{x\}$
    $\tilde{c}_x \leftarrow c_x$
return $\tilde{T}, \tilde{c}$

We prove the privacy guarantees in three steps. First, we show that changing either a single counter or all counters by 1 does not change the output distribution significantly (Corollary 7). This assumes that, for neighboring inputs, the set of stored elements is exactly the same. By Lemma 5, we have that the difference between the sets of stored keys is small and the corresponding counters are ≤ 1. Relying on the thresholding, we bound the probability of outputting one of these keys (Lemma 8). Finally, we combine these two lemmas to show that the privacy guarantees hold for all cases (we do this in Lemma 9).

Lemma 6. *Let us have $x, x' \in \mathbb{R}^k$ such that one of the following three cases holds*

1. *$\exists i \in [k]$ such that $|x_i - x'_i| = 1$ and $x_j = x'_j$ for all $j \neq i$.*
2. *$x_i = x'_i - 1$ for all $i \in [k]$.*
3. *$x_i = x'_i + 1$ for all $i \in [k]$.*

Then we have for any measurable set Z that

$$\Pr[x + \text{Laplace}(1/\varepsilon)^{\otimes k} + \text{Laplace}(1/\varepsilon)\mathbf{1}^k \in Z] \leq e^\varepsilon \Pr[x' + \text{Laplace}(1/\varepsilon)^{\otimes k} + \text{Laplace}(1/\varepsilon)\mathbf{1}^k \in Z]$$

Proof. For the sake of brevity, we let $L = \text{Laplace}(1/\varepsilon)$. Throughout the proof, we construct sets by applying a translation to all elements of another set. That is, for any $\phi \in \mathbb{R}^k$ and measurable set Z we define $Z - \phi = \{a \in \mathbb{R}^k | a + \phi \in Z\}$. We first focus on the simpler case (1). It holds by the law of total expectation that

$$\begin{aligned}\Pr[x + L^{\otimes k} + L\mathbf{1}^k \in Z] &= \\ \mathbb{E}_{N\sim L}\big[\Pr[L^{\otimes k} \in Z - x - N\mathbf{1}^k | N]\big] &\leq \\ e^\varepsilon \mathbb{E}_{N\sim L}\big[\Pr[L^{\otimes k} \in Z - x' - N\mathbf{1}^k | N]\big] &= \\ e^\varepsilon \Pr[x' + L^{\otimes k} + L\mathbf{1}^k \in Z]&\end{aligned}$$

where to prove the inequality, we used that for any measurable set A, it holds

$$\Pr[L^{\otimes k} \in A] \leq e^\varepsilon \Pr[L^{\otimes k} \in A - \phi]$$

for any $\phi \in \mathbb{R}^k$ with $\|\phi\|_1 \leq 1$ (see [8]). Specifically, we have set $A = Z - x - N\mathbf{1}^k$ and $\phi = x - x'$ such that $\|\phi\|_1 = 1$.

We now focus on the cases (2), (3). We will prove below that for x, x' satisfying one of the conditions (2), (3) and for any measurable A, Z and $N_1 \sim L^{\otimes k}$, it holds

$$\begin{aligned}\Pr[x + N_1 + L\mathbf{1}^k \in Z | N_1 \in A] &\\ \leq e^\varepsilon \Pr[x' + N_1 + L\mathbf{1}^k \in Z | N_1 \in A]&\end{aligned}$$

This allows us to argue like above:

$$\begin{aligned}\Pr[x + L^{\otimes k} + L\mathbf{1}^k \in Z] &= \\ \mathbb{E}_{N_1\sim L^{\otimes k}}\big[\Pr[x + N_1 + L\mathbf{1}^k \in Z | N_1]\big] &\leq \\ e^\varepsilon \mathbb{E}_{N_1\sim L^{\otimes k}}\big[\Pr[x' + N_1 + L\mathbf{1}^k \in Z | N_1]\big] &= \\ e^\varepsilon \Pr[x' + L^{\otimes k} + L\mathbf{1}^k \in Z]&\end{aligned}$$

which would conclude the proof. Let $g : \mathbb{R} \to \mathbb{R}^k$ be the function $g(a) = a\mathbf{1}^k$ and define $g^{-1}(B) = \{a \in \mathbb{R} | g(a) \in B\}$ and note that g is measurable. We focus on the case (2); the same argument works for (3) as we discuss below. It holds

$$\begin{aligned}\Pr[x + N_1 + L\mathbf{1}^k \in Z | N_1 \in A] &= \\ \Pr[L\mathbf{1}^k \in Z - x - N_1 | N_1 \in A] &= \\ \Pr[L \in g^{-1}(Z - x - N_1) | N_1 \in A] &= \\ \Pr[L \in g^{-1}(Z - x' - \mathbf{1}^k - N_1) | N_1 \in A] &= \\ \Pr[L \in g^{-1}(Z - x' - N_1) - 1 | N_1 \in A] &\leq \\ e^\varepsilon \Pr[L \in g^{-1}(Z - x' - N_1) | N_1 \in A] &= \\ e^\varepsilon \Pr[L\mathbf{1}^k \in Z - x' - N_1 | N_1 \in A] &= \\ e^\varepsilon \Pr[x' + N_1 + L\mathbf{1}^k \in Z | N_1 \in A].&\end{aligned}$$

To prove the inequality, we again used the standard result that for any measurable A, $\Pr[L \in A] \leq e^\varepsilon \Pr[L \in A - 1]$ holds. The same holds for $A + 1$; this allows us to use the exact same argument in case (3), in which the proof is the same except that -1 on lines 4,5 of the manipulations is replaced by $+1$. □

Corollary 7. *Let T, c and T', c' be two sketches such that $T = T'$ and one of following holds:*

1. *$\exists i \in T$ such that $|c_i - c'_i| = 1$ and $c_j = c'_j$ for all $j \neq i$.*
2. *$c_i = c'_i - 1$ for all $i \in T$.*
3. *$c_i = c'_i + 1$ for all $i \in T$.*

Then for any measurable set of outputs Z, we have:

$$\Pr[\text{PMG}(T, c) \in Z] \leq e^\varepsilon \Pr[\text{PMG}(T', c') \in Z]$$

Proof. Consider first a modified algorithm PMG′ that does not perform the thresholding: that is, if we remove the condition on line 5. It can be easily seen that PMG′ only takes the vector c and releases $c + \text{Laplace}(1/\varepsilon)^{\otimes k} + \text{Laplace}(1/\varepsilon)\mathbf{1}^k$. We have just shown in Lemma 6 that this means that for any measurable Z',

$$\Pr[\text{PMG}'(T,c) \in Z'] \leq e^{\varepsilon} \Pr[\text{PMG}'(T',c') \in Z'].$$

Let $\tau(x) = x$ for $x \geq 1 + 2\ln(3/\delta)/\varepsilon$ and 0 otherwise. Since $\text{PMG}(T,c) = \tau(\text{PMG}'(T,c))$, it then holds

$$\Pr[\text{PMG}(T,c) \in Z] = \Pr[\text{PMG}'(T,c) \in \tau^{-1}(Z)] \leq$$
$$e^{\varepsilon} \Pr[\text{PMG}'(T',c') \in \tau^{-1}(Z)] = e^{\varepsilon} \Pr[\text{PMG}(T',c') \in Z]$$

as we wanted to show. □

Next, we bound the effect on the output distribution from keys that differ between sketches by δ.

Lemma 8. *Let T, c and T', c' be two sketches of size k and let $\hat{T} = T \cap T'$. If we have that $|\hat{T}| \geq k - 2$, $c_i = c'_i$ for all $i \in \hat{T}$, and for all $x \notin \hat{T}$, it holds $c_x, c'_x \leq 1$. Then for any measurable set Z, it holds*

$$\Pr[\text{PMG}(T,c) \in Z] \leq \Pr[\text{PMG}(T',c') \in Z] + \delta$$

Proof. Let $\text{PMG}'(T,c)$ denote a mechanism that executes $\text{PMG}(T,c)$ and performs post-processing by discarding any elements not in $\hat{T}$. It is easy to see that (a) $\Pr[\text{PMG}'(T,c) \in Z] = \Pr[\text{PMG}'(T',c') \in Z]$ since the input sketches are identical for all elements in $\hat{T}$. Moreover, for any output $\tilde{T}, \tilde{c} \leftarrow \text{PMG}(T,c)$ for which $\tilde{T} \subseteq \hat{T}$, the post-processing does not affect the output. This gives us the following inequalities: (b) $\Pr[\text{PMG}(T,c) \in Z] \leq \Pr[\text{PMG}'(T,c) \in Z] + \Pr[\tilde{T} \not\subseteq \hat{T}]$ and (c) $\Pr[\text{PMG}'(T',c') \in Z] \leq \Pr[\text{PMG}(T,c) \in Z] + \Pr[\tilde{T}' \not\subseteq \hat{T}]$. Combining equations $(a) - (c)$, we get the inequality $\Pr[\text{PMG}(T,c) \in Z] \leq \Pr[\text{PMG}(T',c') \in Z] + \Pr[\tilde{T} \not\subseteq \hat{T}] + \Pr[\tilde{T}' \not\subseteq \hat{T}]$.

As such, the Lemma holds if $\Pr[\tilde{T} \not\subseteq \hat{T}] + \Pr[\tilde{T}' \not\subseteq \hat{T}] \leq \delta$. That is, it suffices to prove that with probability at most δ any noisy count for elements not in $\hat{T}$ is at least $1 + 2\ln(3/\delta)/\varepsilon$. The noisy count for such a key can only exceed the threshold if one of the two noise samples added to the key is at least $\ln(3/\delta)/\varepsilon$. From Definition 3 we have $\Pr[\text{Laplace}(1/\varepsilon) \geq \ln(3/\delta)/\varepsilon] = \delta/6$. There are at most 4 keys not in $\hat{T}$ which are in $T \cup T'$ and therefore at most 6 noise samples affect the probability of outputting such a key (the 4 individual Laplace noise samples and the 2 global Laplace noise samples, one for each sketch). By a union bound the probability that any of these samples exceeds $\ln(3/\delta)/\varepsilon$ is at most δ. □

We are now ready to prove the privacy guarantee of Algorithm 2.

Lemma 9. *Algorithm 2 is (ε, δ)-differentially private for any k.*

Proof. The Lemma holds if and only if for any pair of neighboring streams $S \sim S'$ and any measurable set Z we have:

$$\Pr[\text{PMG}(T,c) \in Z] \leq e^{\varepsilon} \Pr[\text{PMG}(T',c') \in Z] + \delta,$$

where $T, c \leftarrow \text{MG}(k, S)$ and $T', c' \leftarrow \text{MG}(k, S')$ denotes the non-private sketches for each stream.

We prove the guarantee above using an intermediate sketch that "lies between" T, c and T', c'. The sketch has support T' and we denote the counters as $\hat{c}$. By Lemma 5, we know that $|T \cap T'| \geq k - 2$ and all counters in c and not in $T \cap T'$ are at most 1. We will now come up with some conditions on $\hat{c}$ such that if these conditions hold, the lemma follows. We will then prove the existence of such $\hat{c}$ below. Assume that $\hat{c}_i = c_i$ for all $i \in T \cap T'$ and $\hat{c}_j \leq 1$ for all $j \in T' \setminus T$. Lemma 8 then tells us that

$$\Pr[\text{PMG}(T,c) \in Z] \leq \Pr[\text{PMG}(T',\hat{c}) \in Z] + \delta.$$

Assume also that one of the required cases for Corollary 7 holds between $\hat{c}$ and c'. We have

$$\Pr[\text{PMG}(T',\hat{c}) \in Z] \leq e^{\varepsilon} \Pr[\text{PMG}(T',c') \in Z].$$

Therefore, if such a sketch $T', \hat{c}$ exists for all S and S' the lemma holds since

$$\begin{aligned}\Pr[\text{PMG}(T,c) \in Z] &\leq \Pr[\text{PMG}(T',\hat{c}) \in Z] + \delta \\ &\leq e^{\varepsilon} \Pr[\text{PMG}(T',c') \in Z] + \delta \ .\end{aligned}$$

It remains to prove the existence of $\hat{c}$ such that $\hat{c}_i = c_i$ for all $i \in T \cap T'$ and $\hat{c}_j \leq 1$ for all $j \in T' \setminus T$ and such that one of the conditions $(1) - (3)$ of Corollary 7 holds between $\hat{c}$ and c'. We first consider neighboring streams where S' is obtained by removing an element from S. From Lemma 5 we have two cases to consider. If $c_i = c'_i - 1$ for all $i \in T'$ we simply set $\hat{c} = c$. Recall that we implicitly have $c_i = 0$ for $i \notin T$. Therefore the sketch satisfies the two conditions above since $\hat{c}_i = c_i$ for all $i \in \mathcal{U}$ and condition (2) of Corollary 7 holds. In the other case where $c_i = c'_i + 1$ for exactly one $i \in T$ there are two possibilities. If $i \in T'$ we again set $\hat{c} = c$. When $i \notin T'$ there must exist at least one element $j \in T'$ such that $c'_j = 0$ and $j \notin T$. We set $\hat{c}_j = 1$ and $\hat{c}_i = c'_i$ for all $i \neq j$. In both cases $\hat{c}_i = c_i$ for all $i \in T \cap T'$ and $\hat{c}_j$ is at most one for $j \notin T$. There is exactly one element with a higher count in $\hat{c}$ than c' which means that condition (1) of Corollary 7 holds.

If S is obtained by removing an element from S' the cases from Lemma 5 are flipped. If $c_i - 1 = c'_i$ for all $i \in T$ and $c'_j = 0$ for $j \notin T$ we set $\hat{c}_i = c_i$ if $i \in T$ and $\hat{c}_i = 1$ otherwise. It clearly holds that $\hat{c}_i = c_i$ for all $i \in T \cap T'$ and $\hat{c}_j \leq 1$ for all $j \notin T$. Since $\hat{c}_i = c'_i + 1$ for all $i \in T'$ condition (3) of Corollary 7 holds. Finally, if $c_i + 1 = c'_i$ for exactly one $i \in T'$ we simply set $\hat{c} = c$. $\hat{c}_i = c_i$ clearly holds for all $i \in T \cap T'$, $\hat{c}_j = 0$ for all $j \notin T$, and condition (1) of Corollary 7 holds between $\hat{c}$ and c'. □

Next, we analyze the error compared to the non-private sketch. We state the error in terms of the largest error among all elements of the sketch. Recall that we implicitly say that the count is zero for any element not in the sketch.

Lemma 10. *Let $\tilde{T}, \tilde{c} \leftarrow \text{PMG}(T,c)$ denote the output of Algorithm 2 for any sketch T, c with $|T| = k$. Then with probability at least $1 - \beta$ we have*

$$\tilde{c}_x \in \left[c_x - \frac{2\ln\left(\frac{k+1}{\beta}\right)}{\varepsilon} - 1 - \frac{2\ln\left(3/\delta\right)}{\varepsilon}, c_x + \frac{2\ln\left(\frac{k+1}{\beta}\right)}{\varepsilon} \right]$$

for all $x \in T$ and $\tilde{c}_x = 0$ for all $x \notin T$.

Proof. The two sources of error are the noise samples and the thresholding step. We begin with a simple bound on the absolute value of the Laplace distribution.

$$\Pr\left[|\text{Laplace}(1/\varepsilon)| \geq \frac{\ln((k+1)/\beta)}{\varepsilon}\right] =$$
$$2 \cdot \Pr\left[\text{Laplace}(1/\varepsilon) \leq -\frac{\ln((k+1)/\beta)}{\varepsilon}\right] = \beta/(k+1) \ .$$

Since $k+1$ samples are drawn we know by a union bound that the absolute value of all samples is bounded by $\ln((k+1)/\beta)/\varepsilon$ with probability at least $1-\beta$. As such the absolute error from the Laplace samples is at most $2\ln((k+1)/\beta)/\varepsilon$ for all $x \in T$ since two samples are added to each count. Removing noisy counts below the threshold potentially adds an additional error of at most $1 + 2\ln(3/\delta)/\varepsilon$. It is easy to see that $\tilde{c}_x = 0$ for all $x \notin T$ since the algorithm never outputs any such elements. □

Theorem 11. PMG(k, S) *satisfies* (ε, δ)*-differential privacy. Let* $f(x)$ *denote the frequency of any element* $x \in \mathcal{U}$ *in* S *and let* $\hat{f}(x)$ *denote the estimated frequency of* x *from the output of* PMG(k, S)*. For any* x *with* $f(x) = 0$ *we have* $\hat{f}(x) = 0$ *and with probability at least* $1 - \beta$ *we have for all* $x \in \mathcal{U}$ *that*

$$\hat{f}(x) \in \left[f(x) - \frac{2\ln\left(\frac{k+1}{\beta}\right)}{\varepsilon} - 1 - \frac{2\ln(3/\delta)}{\varepsilon} - \frac{|S|}{k+1}, f(x) + \frac{2\ln\left(\frac{k+1}{\beta}\right)}{\varepsilon}\right]$$

Moreover, the algorithm outputs all x*, such that* $\hat{f}(x) > 0$ *and there are at most* k *such elements.* PMG(k, S) *uses* $2k$ *words of memory. For any fixed* $x \in \mathcal{U}$*, the mean squared error is* $\mathbb{E}[(\hat{f}(x) - f(x))^2] \leq 3\left(1 + \frac{2+2\ln(3/\delta)}{\varepsilon} + \frac{|S|}{k+1}\right)^2$.

Proof. The space complexity is clearly as claimed, as we are storing at any time at most k items and counters. We focus on proving privacy and correctness.

If $f(x) = 0$ we know that $x \notin T$ where T is the keyset after running Algorithm 1. Since Algorithm 2 outputs a subset of T we have $\hat{f}(x) = 0$. The first part of the Theorem follows directly from Fact 4 and Lemmas 9 and 10.

We now bound the mean squared error. There are three sources of error. Let r_1 be the error coming from the Laplace noise, r_2 from the thresholding, and r_3 the error made by the MG sketch. Then

$$\begin{aligned}\mathbb{E}[(\hat{f}(x) - f(x))^2] =& \mathbb{E}[(r_1 + r_2 + r_3)^2] \\ \leq& 3(\mathbb{E}[r_1^2] + \mathbb{E}[r_2^2] + \mathbb{E}[r_3^2])\end{aligned}$$

by equivalence of norms (for any dimension n vector v, $\|v\|_1 \leq \sqrt{n}\|v\|_2$). The errors r_2, r_3 are deterministically bounded $r_2 \leq 1+2\ln(3/\delta)/\varepsilon$ and $r_3 \leq |S|/(k+1)$. $\mathbb{E}[r_1^2]$ is the variance of the Laplace noise; we added two independent noises each with scale $1/\varepsilon$ and thus variance $2/\varepsilon^2$ for a total variance of $4/\varepsilon^2$. This finishes the proof. □

5.1 Privatizing standard versions of MG

The privacy of our mechanism as presented in Algorithm 2 relies on our variant of the Misra-Gries algorithm. Our sketch can contain elements with a count of zero. However, elements with a count of zero are removed in the standard version of the sketch. As such, sketches for neighboring datasets can differ for up to k keys if one sketch stores k elements with a count of 1 and the other sketch is empty. It is easy to change Algorithm 2 to handle these implementations. We simply increase the threshold to $1 + 2\ln\left(\frac{k+1}{2\delta}\right)/\varepsilon$ since the probability of outputting any of the k elements with a count of 1 is bounded by δ.

5.2 Tips for practitioners

Here we discuss some technical details to keep in mind when implementing our mechanism.

The output of the Misra-Gries algorithm is an associative array. In Algorithm 2 we add appropriate noise such that the associative array can be released under differential privacy. However, for some implementations of associative arrays such as hash tables the order in which keys are added affects the data structure. Using such an implementation naively violates differential privacy but it is easily solved either by outputting a random permutation of the key-value pairs or using a fixed order e.g. sorted by key.

We present our mechanism with noise sampled from the Laplace distribution. However, the distribution is defined for real numbers which cannot be represented on a finite computer. This is a known challenge and precision-based attacks still exist on popular implementations [13]. Since the output of MG is discrete the distribution can be replaced by the Geometric mechanism [12] or one of the alternatives introduced in [2]. Our mechanism would still satisfy differential privacy but it might be necessary to change the threshold in Algorithm 2 slightly to ensure that Lemma 8 still holds. Our proof of Lemma 8 works for the Geometric mechanism from [12] when increasing the threshold to $1 + 2\lceil\ln(6e^\varepsilon/((e^\varepsilon+1)\delta))/\varepsilon\rceil$.

Lastly, it is worth noting that the analysis for Lemma 8 is not tight. We bound the probability of outputting a small key by bounding the value of all relevant samples by $\ln(3/\delta)/\varepsilon$ which is sufficient to guarantee that the sum of any two samples does not exceed $2\ln(3/\delta)/\varepsilon$. This simplifies the proof and presentation significantly however one sample could exceed $\ln(3/\delta)/\varepsilon$ without any pair of samples exceeding $2\ln(3/\delta)/\varepsilon$. A tighter analysis would improve the constant slightly which might matter for practical applications.

6. PRIVATIZING MERGED SKETCHES

In practice, it is often important that we may merge sketches. This is for example commonly used when we have a dataset distributed over many servers. Each dataset consists of multiple streams in this setting, and we want to compute an aggregated sketch over all streams. We say that datasets are neighboring if we can obtain one from the other by removing a single element from one of the streams. If the aggregator is untrusted we must add noise to each sketch before performing any merges. This is the setting in [6] and we can run their merging algorithm. However, since we add noise to each sketch the error scales with the number of sketches. In particular, the error from the thresholding step of Algorithm 2 scales linearly in the number of sketches for worst-case input. In the full version of the paper we show how we can sometimes achieve smaller error in the setting where aggregators are trusted.

Acknowledgment

The authors are supported by the VILLUM Foundation grant 16582. We thank Rasmus Pagh for suggesting this problem and for helpful discussions. We thank Martin Aumüller for his valuable feedback.

7. REFERENCES

[1] M. Aumüller, C. J. Lebeda, and R. Pagh. Representing sparse vectors with differential privacy, low error, optimal space, and fast access. *Journal of Privacy and Confidentiality*, 12(2), Nov. 2022.

[2] V. Balcer and S. Vadhan. Differential privacy on finite computers. *Journal of Privacy and Confidentiality*, 9(2), Sep. 2019.

[3] R. Bassily, K. Nissim, U. Stemmer, and A. Guha Thakurta. Practical locally private heavy hitters. *Advances in Neural Information Processing Systems*, 30, 2017.

[4] J. Böhler and F. Kerschbaum. Secure multi-party computation of differentially private heavy hitters. In Y. Kim, J. Kim, G. Vigna, and E. Shi, editors, *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, pages 2361–2377. ACM, 2021.

[5] P. Bose, E. Kranakis, P. Morin, and Y. Tang. Bounds for frequency estimation of packet streams. In J. F. Sibeyn, editor, *SIROCCO 10: Proceedings of the 10th Internaltional Colloquium on Structural Information Complexity, June 18-20, 2003, Umeå Sweden*, volume 17 of *Proceedings in Informatics*, pages 33–42. Carleton Scientific, 2003.

[6] T.-H. H. Chan, M. Li, E. Shi, and W. Xu. Differentially private continual monitoring of heavy hitters from distributed streams. In *International Symposium on Privacy Enhancing Technologies Symposium*, pages 140–159. Springer, 2012.

[7] G. Cormode, C. M. Procopiuc, D. Srivastava, and T. T. L. Tran. Differentially private summaries for sparse data. In *ICDT*, pages 299–311. ACM, 2012.

[8] C. Dwork, F. McSherry, K. Nissim, and A. Smith. Calibrating noise to sensitivity in private data analysis. pages 265–284, 2006.

[9] C. Dwork and A. Roth. The algorithmic foundations of differential privacy. *Foundations and Trends in Theoretical Computer Science*, 9(3-4):211–407, 2014.

[10] Q. Geng, P. Kairouz, S. Oh, and P. Viswanath. The staircase mechanism in differential privacy. *IEEE Journal of Selected Topics in Signal Processing*, 9(7):1176–1184, 2015.

[11] B. Ghazi, N. Golowich, R. Kumar, R. Pagh, and A. Velingker. On the power of multiple anonymous messages. *IACR Cryptol. ePrint Arch.*, page 1382, 2019.

[12] A. Ghosh, T. Roughgarden, and M. Sundararajan. Universally utility-maximizing privacy mechanisms. *SIAM Journal on Computing*, 41(6):1673–1693, 2012.

[13] S. Haney, D. Desfontaines, L. Hartman, R. Shrestha, and M. Hay. Precision-based attacks and interval refining: how to break, then fix, differential privacy on finite computers. *CoRR*, abs/2207.13793, 2022.

[14] A. Korolova, K. Kenthapadi, N. Mishra, and A. Ntoulas. Releasing search queries and clicks privately. In *WWW*, pages 171–180. ACM, 2009.

[15] J. Misra and D. Gries. Finding repeated elements. *Science of Computer Programming*, 2(2):143–152, 1982.

[16] R. Pagh and M. Thorup. Improved utility analysis of private countsketch. In *Advances in Neural Information Processing Systems*, volume 35, pages 25631–25643, 2022.

[17] F. Zhao, D. Qiao, R. Redberg, D. Agrawal, A. El Abbadi, and Y.-X. Wang. Differentially private linear sketches: Efficient implementations and applications. In *Advances in Neural Information Processing Systems*, volume 35, pages 12691–12704, 2022.

Technical Perspective: Allocating Isolation Levels to Transactions in a Multiversion Setting

Alan D. Fekete
University of Sydney
alan.fekete@sydney.edu.au

Among the ways a database management system adds value, is the transaction abstraction, where the application coder can group together multiple data accesses that collectively perform one meaningful real-world activity. The platform will provide the "ACID" properties (atomic, consistent, isolated and durable) so the whole transaction happens like a single event. The mechanisms that allow this appearance include crash recovery and rollback (usually based on log entries) and concurrency control (typically involving locks).

To capture the essential goal of concurrency control, we consider whether a given execution of the system is "serializable", which means that there is another execution which is not-interleaved at all, and has the same outcome for both final state of the data, and the values observed in each transaction. In a serializable execution, any integrity property is guaranteed to hold at the end, even if it is not explicitly enforced by the platform, as long as each program running alone would preserve that integrity. The ideal is that every execution of the platform will be serializable.

An early mechanism for concurrency control was 2-Phase Locking, based on an update-in-place approach to data, and taking exclusive locks (for data modification) or shareable locks (for reads) and holding them till the end of the transaction. Many recent platforms instead use *multiversion* mechanisms where some accesses work on an older version of a data item, rather than the latest version; this can allow better performance because a read might proceed even before a concurrent writer has completed.

From early days of database technology, platforms have offered in the API a command that the application can invoke to SET TRANSACTION ISOLATION LEVEL to SERIALIZABLE, REPEATABLE READ, or READ COMMITTED. Gray et al [1] introduced Read Committed isolation as a variation on locking where sharable locks are held only during a read access, rather than till the end of the transaction as in traditional serializable 2-Phase Locking. Doing this may improve performance, but the data integrity could be at risk. There are also variations on multi-version concurrency control which give less isolation than serializability, though the system behaviors allowed at a given level are not exactly the same as what happens with the traditional locking-based ways.

The expectation of weak isolation, is that the application coder will decide which isolation level each transaction should use, based on their understanding of what is safe to do. Making this decision in a reasoned way, for the multiversion isolation mechanisms, is what is tackled by the recent work of Vandevoort, Ketsman and Neven. The way this work defines whether an allocation is safe to use for particular transactions (called *robust*), is to ask that every possible execution of those transactions when each is run with the chosen isolation level, will be serializable. Determining whether a given allocation is robust naturally leads to the question of finding the best allocation which is robust.

The work of Vandervoort et al is done in the style of theoretical computer science; there is a focus on precise definitions, and knowing whether a particular calculation takes time which is polynomial in the size of the input, or worse. There are as always, some simplifications in the theory, which keep the mathematics and notation manageable. This work extends a long tradition of theory applied to concurrency control [2]. Among the notable contributions of the paper by Vandervoort et al, is to capture what executions are possible for a given allocation. This is important because the isolation levels are typically described as if all transactions have the given isolation. Understanding how an execution can proceed with mixed isolation, requires careful looking at the implementation, and then abstracting out the interactions to state the key properties. The paper also offers a very nice theorem, that shows that if an allocation allows an interleaving of the transactions which is non-serializable, then there exists an interleaving of a particularly simple form, which the authors call *split schedule*. This allows one to test for robustness by checking far fewer interleavings, and is the heart of how this paper gets polynomial-time algorithms.

1. REFERENCES

[1] J. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger. Granularity of locks and degrees of consistency in a shared data base. In *Modelling in Data Base Management Systems, Proceeding of the IFIP Working Conference on Modelling in Data Base Management Systems, Freudenstadt, Germany, January 5-8, 1976*, pages 365–394, 1976. available at http://jimgray.azurewebsites.net/JimGrayPublications.htm.

[2] G. Weikum and G. Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann, 2002.

Allocating Isolation Levels to Transactions in a Multiversion Setting

Brecht Vandevoort
UHasselt, Data Science Institute, ACSL
brecht.vandevoort@uhasselt.be

Bas Ketsman
Vrije Universiteit Brussel
bas.ketsman@vub.be

Frank Neven
UHasselt, Data Science Institute, ACSL
frank.neven@uhasselt.be

ABSTRACT

A serializable concurrency control mechanism ensures consistency for OLTP systems at the expense of a reduced transaction throughput. A DBMS therefore usually offers the possibility to allocate lower isolation levels for some transactions when it is safe to do so. However, such trading of consistency for efficiency does not come with any safety guarantees. In this paper, we study the mixed robustness problem which asks whether, for a given set of transactions and a given allocation of isolation levels, every possible interleaved execution of those transactions that is allowed under the provided allocation is always serializable. That is, whether the given allocation is indeed safe. While robustness has already been studied in the literature for the homogeneous setting where all transactions are allocated the same isolation level, the heterogeneous setting that we consider in this paper, despite its practical relevance, has largely been ignored. We focus on multiversion concurrency control and consider the isolation levels that are available in Postgres and Oracle: read committed (RC), snapshot isolation (SI) and serializable snapshot isolation (SSI). We show that the mixed robustness problem can be decided in polynomial time. In addition, we provide a polynomial time algorithm for computing the optimal robust allocation for a given set of transactions, prioritizing lower over higher isolation levels. The present results therefore establish the groundwork to automate isolation level allocation within existing databases supporting multiversion concurrency control.

1. INTRODUCTION

The majority of relational database systems offer a range of isolation levels, the highest of which is serializability ensuring what is considered as perfect isolation. This allows users to trade off isolation guarantees for better performance. Executing transactions concurrently at weaker degrees of isolation does carry some risk as it can result in specific anomalies. However, there are situations when a group of transactions can be executed at an isolation level lower than serializability without causing any errors. In this way, we get the higher isolation guarantees of serializability for free in exchange for a lower isolation level, which is typically implementable with a less expensive concurrency control mechanism. This formal property is called *robustness* [13, 19, 20]: a set of transactions $\mathcal{T}$ is called robust against a given isolation level if every possible interleaving of the transactions in $\mathcal{T}$ that is allowed under the specified isolation level is serializable. There is a famous example that is part of database folklore: the TPC-C benchmark [24] is robust against Snapshot Isolation (SI), so there is no need to run a stronger, and more expensive, concurrency control algorithm than SI if the workload is just TPC-C. This has played a role in the incorrect choice of SI as the general concurrency control algorithm for isolation level Serializable in Oracle and PostgreSQL (before version 9.1, cf. [20]).

The robustness problem received quite a bit of attention in the literature and can be classified in terms of the considered isolation levels: lower isolation levels like (multiversion) Read Committed (RC) [6,22,25,26], Snapshot Isolation (SI) [4, 10, 19, 20], and higher isolation levels [11, 13, 16, 18]. The far majority of this work focused on a *homogeneous* setting where all transactions are allocated the *same* isolation level. So, when a workload is robust against an isolation level, all transactions can be executed under this isolation level and benefit from the speedup offered by the cheaper concurrency control algorithm and the guarantee that the resulting execution will always be serializable. When a workload is not robust against an isolation level, robustness can still be achieved by modifying the transaction programs [3–6, 20, 25] or using an external lock manager [3, 6, 7]. The downside of these solutions is that they require altering transactions or require drastic changes to the underlying database implementation.

In this paper, we are interested in solutions that refrain from modifying transactions and can be readily used on top of a DBMS without changing any of the database internals. The solution lies within the capabilities of the DBMS itself. Indeed, in practice, an isolation level is not set on the level of the database or even the application but can be specified on the level of an individual transaction. So, a third option for making a transaction workload robust is to allocate problematic transactions to higher isolation levels. That is, by considering *heterogeneous* or *mixed* allocations where individual transactions can be mapped to different

isolation levels. Such an approach requires a solution for two research challenges as discussed next: the *robustness problem* and the *allocation problem*. To this end, let $\mathcal{T}$ be a set of transactions, $\mathcal{I}$ a class of isolation levels and $\mathcal{A}$ an allocation (mapping each $T \in \mathcal{T}$ to an isolation level in $\mathcal{I}$). Then define the following problems:

- The **robustness problem for $\mathcal{I}$**: Is every concurrent execution of transactions in $\mathcal{T}$ that is allowed under $\mathcal{A}$, conflict-serializable?
- The **allocation problem for $\mathcal{I}$**: Compute an optimal robust allocation for $\mathcal{T}$ over $\mathcal{I}$ (when it exists).

In order to increase transaction throughput, weaker isolation levels, which are often less strict and permit higher concurrency, are favored over stricter isolation levels which generally limit concurrency.[1] We then say that a robust allocation is *optimal* when no higher isolation level can be exchanged for a weaker one without breaking robustness. A seminal result in this context is that of Fekete [19] who provided polynomial time algorithms for the robustness and the allocation problem for the setting where $\mathcal{I}$ consists of the isolation levels SI and strict two-phase locking (S2PL). More specifically, when $\mathcal{T}$ is not robust against SI, a minimal number of transactions can be found that need to be run under S2PL to make the workload robust.

In the present work, we address the robustness and allocation problem for a wider range of isolation levels: RC, SI, and Serializable Snapshot Isolation (SSI) [14, 23]. These classes are particularly relevant for the following reasons: RC is often configured by default [9]; SI remains the highest possible isolation level in some database systems like Oracle and is well-studied (e.g, [4, 10, 13, 16–21]; and, SSI effectively guarantees serializability. Furthermore, $\{\text{RC}, \text{SI}, \text{SSI}\}$ is the class of isolation levels available in Postgres, while $\{\text{RC}, \text{SI}\}$ are those available in Oracle. We see our results as a significant step towards automating isolation level allocation on top of existing databases. Indeed, we obtain that for $\{\text{RC}, \text{SI}, \text{SSI}\}$ an optimal robust allocation can always be found in polynomial time. As $\{\text{RC}, \text{SI}\}$ does not include a serializable isolation level, a robust allocation does not always exist. However, the results in this paper show that the existence of a robust allocation for $\{\text{RC}, \text{SI}\}$ can be decided in polynomial time, and when a robust allocation exists, an optimal one can be found.

The main technical contribution of this paper is Theorem 3.2 which shows that non-robustness against an allocation for the isolation levels $\{\text{RC}, \text{SI}, \text{SSI}\}$ can be characterized in terms of the existence of a counterexample schedule of a very specific form that we refer to as a multiversion split schedule. Such split schedules have been used before in the homogeneous setting where all transactions in a workload are assigned to the same isolation level [19, 22, 25]. Generally, a split schedule is of the following form: one transaction is split in two (hence, the name) and some other transactions are placed between these two parts in a serial fashion where both the splitted and the intermediate serial transactions satisfy some additional requirements. All remaining transactions (if any) are appended after the splitted transaction, again, in a serial fashion. We refer to Figure 1 for

[1]Indeed, Vandevoort et al. [25] have shown that when contention increases, RC outperforms SI w.r.t. transaction throughput.

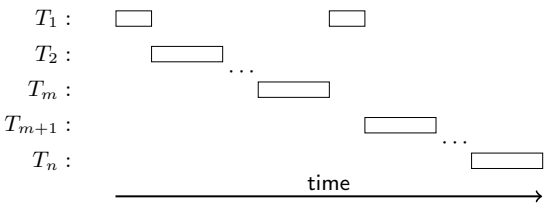


Figure 1: Abstract representation of a multi-version split schedule where T_1 is the splitted transaction.

the general structure of a split schedule. The split schedules used in the cited papers all differ in the additional requirements. When these additional requirements are simple, a direct enumeration of all possible split schedules can be avoided and replaced by a more efficient polynomial time algorithm [22, 25]. In some cases, however, finding a counterexample split schedule is NP-complete [22] or even undecidable [26]. In the present paper, we consider mixed allocations where different transactions can be allocated to different isolation levels. The corresponding split schedule is consequentially more involved as it needs to take interrelationships between multiple isolation levels into account. We show in Theorem 3.3 that a counterexample split schedule can still be efficiently constructed.

The contributions of this paper can be summarized as follows:

1. We provide a formal framework to reason on robustness in the presence of mixed allocations of isolation levels. In particular, we formally define what it means for a schedule to be allowed under a (mixed) allocation w.r.t. $\{\text{RC}, \text{SI}, \text{SSI}\}$ (cf., Definition 2.4). Even though these definitions are an abstraction, they are consistent with mixed allocations as they are applied within Postgress and Oracle.
2. We characterize non-robustness for allocations over $\{\text{RC}, \text{SI}, \text{SSI}\}$ in terms of the existence of a multiversion split-schedule.
3. We provide a polynomial time decision procedure for robustness against an allocation over $\{\text{RC}, \text{SI}, \text{SSI}\}$.
4. We show that there is always a unique optimal robust allocation over $\{\text{RC}, \text{SI}, \text{SSI}\}$ and we provide a polynomial time algorithm for computing it.
5. We show that is decidable in polynomial time whether there exists a robust allocation over $\{\text{RC}, \text{SI}\}$ for a given set of transactions. Furthermore, when a robust allocation exists, an optimal one can be found in polynomial time as well.

Outline. This paper is structured as follows. We introduce the necessary definitions in Section 2. We consider the robustness and allocation problem for $\{\text{RC}, \text{SI}, \text{SSI}\}$ in Section 3 and Section 4, respectively. We consider robustness and allocation for $\{\text{RC}, \text{SI}\}$ in Section 5. We discuss related work in Section 6. We conclude in Section 7.

This paper is a shortened version of the PODS 2023 paper [27] where the definition of a multiversion split schedule (Definition 3.1) has been simplified and a more elaborate running example has been added.

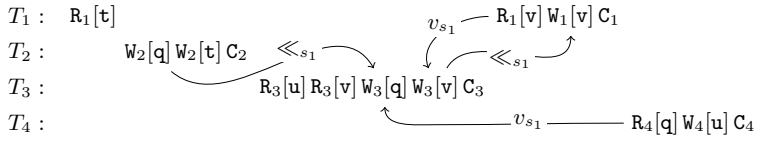


Figure 2: A single version schedule s_1 for $\mathcal{T}_{\text{ex}}$ with v_{s_1} and $\ll_{s_1}$ represented through arrows. The special operation op_0 and all arrows involving op_0 are omitted.

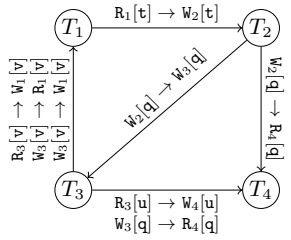


Figure 3: Serialization graph $SeG(s_1)$.

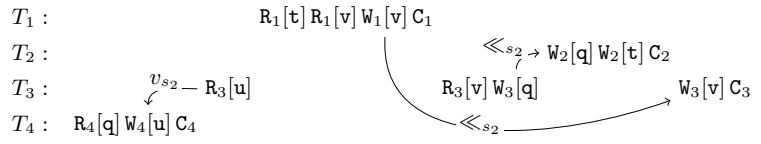


Figure 4: A schedule s_2 for $\mathcal{T}_{\text{ex}}$ with v_{s_2} and $\ll_{s_2}$ represented through arrows. The special operation op_0 and all arrows involving op_0 are omitted.

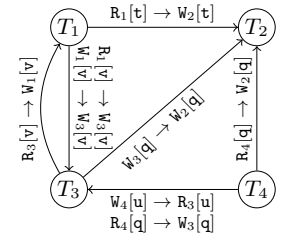


Figure 5: Serialization graph $SeG(s_2)$.

2. DEFINITIONS

2.1 Transactions and Schedules

We fix an infinite set of objects **Obj**. For an object $\mathtt{t} \in$ **Obj**, we denote by $\mathtt{R}[\mathtt{t}]$ a *read* operation on $\mathtt{t}$ and by $\mathtt{W}[\mathtt{t}]$ a *write* operation on $\mathtt{t}$. We also assume a special *commit* operation denoted by $\mathtt{C}$. A *transaction* T over **Obj** is a sequence of read and write operations on objects in **Obj** followed by a commit. In the sequel, we leave the set of objects **Obj** implicit when it is clear from the context and just say transaction rather than transaction over **Obj**.

Formally, we model a transaction as a linear order $(T, \leq_T)$, where T is the set of (read, write and commit) operations occurring in the transaction and $\leq_T$ encodes the ordering of the operations. As usual, we use $<_T$ to denote the strict ordering. For a transaction T, we use *first*(T) to refer to the first operation in T.

When considering a set $\mathcal{T}$ of transactions, we assume that every transaction in the set has a unique id i and write T_i to make this id explicit. Similarly, to distinguish the operations of different transactions, we add this id as a subscript to the operation. That is, we write $\mathtt{W}_i[\mathtt{t}]$ and $\mathtt{R}_i[\mathtt{t}]$ to denote a $\mathtt{W}[\mathtt{t}]$ and $\mathtt{R}[\mathtt{t}]$ occurring in transaction T_i; similarly $\mathtt{C}_i$ denotes the commit operation in transaction T_i. This convention is consistent with the literature (see, *e.g.* [12,19]). To avoid ambiguity of notation, we assume that a transaction performs at most one write and one read operation per object. The latter is a common assumption (see, *e.g.* [19]). All our results carry over to the more general setting in which multiple writes and reads per object are allowed.

As a running example, we define the set of transactions $\mathcal{T}_{\text{ex}} = \{T_1, T_2, T_3, T_4\}$ over four different objects $\mathtt{t}$, $\mathtt{u}$, $\mathtt{v}$, and $\mathtt{q}$ as follows:

- $T_1 = \mathtt{R}_1[\mathtt{t}]\,\mathtt{R}_1[\mathtt{v}]\,\mathtt{W}_1[\mathtt{v}]\,\mathtt{C}_1$;
- $T_2 = \mathtt{W}_2[\mathtt{q}]\,\mathtt{W}_2[\mathtt{t}]\,\mathtt{C}_2$;
- $T_3 = \mathtt{R}_3[\mathtt{u}]\,\mathtt{R}_3[\mathtt{v}]\,\mathtt{W}_3[\mathtt{q}]\,\mathtt{W}_3[\mathtt{v}]\,\mathtt{C}_3$; and
- $T_4 = \mathtt{R}_4[\mathtt{q}]\,\mathtt{W}_4[\mathtt{u}]\,\mathtt{C}_4$.

A *(multiversion) schedule* s over a set $\mathcal{T}$ of transactions is a tuple $(O_s, \leq_s, \ll_s, v_s)$ where

- O_s is the set containing all operations of transactions in $\mathcal{T}$ as well as a special operation op_0 conceptually writing the initial versions of all existing objects,
- $\leq_s$ encodes the ordering of these operations,
- $\ll_s$ is a *version order* providing for each object $\mathtt{t}$ a total order over all write operations on $\mathtt{t}$ occurring in s, and,
- v_s is a *version function* mapping each read operation a in s to either op_0 or to a write operation in s.

We require that $op_0 \leq_s a$ for every operation $a \in O_s$, $op_0 \ll_s a$ for every write operation $a \in O_s$, and that $a <_T b$ implies $a <_s b$ for every $T \in \mathcal{T}$ and every $a, b \in T$. We furthermore require that for every read operation a, $v_s(a) <_s a$ and, if $v_s(a) \neq op_0$, then the operation $v_s(a)$ is on the same object as a. Intuitively, op_0 indicates the start of the schedule, the order of operations in s is consistent with the order of operations in every transaction $T \in \mathcal{T}$, and the version function maps each read operation a to the operation that wrote the version observed by a. If $v_s(a)$ is op_0, then a observes the initial version of this object. The version order $\ll_s$ represents the order in which different versions of an object are installed in the database. For a pair of write operations on the same object, this version order does not necessarily coincide with $\leq_s$. For example, under RC and SI the version order is based on the commit order instead.

Continuing our running example, Figure 2 and Figure 4 illustrate two schedules s_1 and s_2 over $\mathcal{T}_{\text{ex}}$. The version order $\ll$ as well as the version function v are represented as arrows. The special operation op_0 together with its arrows, is omitted in these figures to improve readability. For example, $v_{s_1}(\mathtt{R}_1[\mathtt{v}]) = \mathtt{W}_3[\mathtt{v}]$ in schedule s_1, since $\mathtt{R}_1[\mathtt{v}]$ reads the version of $\mathtt{v}$ written by $\mathtt{W}_3[\mathtt{v}]$. In s_2 on the other hand, we

have $v_{s_2}(\mathtt{R}_1[\mathtt{v}]) = op_0$, since $\mathtt{R}_1[\mathtt{v}]$ reads the initial version of $\mathtt{v}$. Furthermore, the read operation $\mathtt{R}_3[\mathtt{v}]$ reads the initial version of $\mathtt{v}$ in schedule s_2 instead of the version written by T_1, even though T_1 commits before $\mathtt{R}_3[\mathtt{v}]$ in s_2.

We say that a schedule s is a *single version schedule* if $\ll_s$ is compatible with $\leq_s$ and every read operation always reads the last written version of the object. Formally, for each pair of write operations a and b on the same object, $a \ll_s b$ iff $a <_s b$, and for every read operation a there is no write operation c on the same object as a with $v_s(a) <_s c <_s a$. For example, s_1 in Figure 2 is a single version schedule, while s_2 in Figure 4 is not as $op_0 = v_{s_2}(\mathtt{R}_3[\mathtt{v}]) <_{s_2} \mathtt{W}_1[\mathtt{v}] <_{s_2} \mathtt{R}_3[\mathtt{v}]$. A single version schedule over a set of transactions $\mathcal{T}$ is *single version serial* if its transactions are not interleaved with operations from other transactions. That is, for every $a, b, c \in O_s$ with $a <_s b <_s c$ and $a, c \in T$ implies $b \in T$ for every $T \in \mathcal{T}$.

The absence of aborts in our definition of schedule is consistent with the common assumption [13, 19] that an underlying recovery mechanism will rollback aborted transactions. We only consider isolation levels that only read committed versions. Therefore there will never be cascading aborts.

2.2 Conflict Serializability

Two operations a_j and b_i from different transactions T_j and T_i in a set of transactions $\mathcal{T}$ are *conflicting* if they are on the same object $\mathtt{t}$ and at least one of them is a write. In this case, we furthermore say that b_i is *ww-conflicting* (respectively, *wr-conflicting*) with a_j if b_i is a write operation and a_j is a write operation (respectively, a read operation); and b_i is *rw-conflicting* with a_j if b_i is a read operation and a_j is a write operation. Furthermore, commit operations and the special operation op_0 never conflict with any other operation. When b_i and a_j are conflicting operations in $\mathcal{T}$, we say that a_j *depends on* b_i in a schedule s over $\mathcal{T}$, denoted $b_i \rightarrow_s a_j$ if:

- *(ww-dependency)* b_i is ww-conflicting with a_j and $b_i \ll_s a_j$; or,
- *(wr-dependency)* b_i is wr-conflicting with a_j and $b_i = v_s(a_j)$ or $b_i \ll_s v_s(a_j)$; or,
- *(rw-antidependency)* b_i is rw-conflicting with a_j and $v_s(b_i) \ll_s a_j$.

Intuitively, a ww-dependency from b_i to a_j implies that a_j writes a version of an object that is installed after the version written by b_i. An wr-dependency from b_i to a_j implies that b_i either writes the version observed by a_j, or it writes a version that is installed before the version observed by a_j. A rw-antidependency from b_i to a_j implies that b_i observes a version installed before the version written by a_j. For example, the dependencies $\mathtt{W}_3[\mathtt{v}] \rightarrow_{s_1} \mathtt{W}_1[\mathtt{v}]$, $\mathtt{W}_3[\mathtt{v}] \rightarrow_{s_1} \mathtt{R}_1[\mathtt{v}]$ and $\mathtt{R}_3[\mathtt{v}] \rightarrow_{s_1} \mathtt{W}_1[\mathtt{v}]$ are respectively a ww-dependency, a wr-dependency and a rw-antidependency in schedule s_1 presented in Figure 2.

Two schedules s and s' are *conflict-equivalent* if they are over the same set $\mathcal{T}$ of transactions and for every pair of conflicting operations a_j and b_i, $b_i \rightarrow_s a_j$ iff $b_i \rightarrow_{s'} a_j$.

Definition 2.1. *A schedule s is* conflict-serializable *if it is conflict-equivalent to a single version serial schedule.*

A *serialization graph* $SeG(s)$ for schedule s over a set of transactions $\mathcal{T}$ is the graph whose nodes are the transactions in $\mathcal{T}$ and where there is an edge from T_i to T_j if T_j has an operation a_j that depends on an operation b_i in T_i, thus with $b_i \rightarrow_s a_j$.

Theorem 2.2 (implied by [2]). *A schedule s is conflict-serializable iff $SeG(s)$ is acyclic.*

Figure 3 and 5 visualize the serialization graphs $SeG(s_1)$ and $SeG(s_2)$ for the schedules s_1 and s_2 in Figure 2 and 4, respectively. For illustration purposes, each edge is labelled with the corresponding dependencies. Since $SeG(s_1)$ and $SeG(s_2)$ are not acyclic, both schedules are not conflict-serializable.

2.3 Isolation Levels

Let $\mathcal{I}$ be a class of isolation levels. An *$\mathcal{I}$-allocation* $\mathcal{A}$ for a set of transactions $\mathcal{T}$ is a function mapping each transaction $T \in \mathcal{T}$ onto an isolation level $\mathcal{A}(T) \in \mathcal{I}$. When $\mathcal{I}$ is not important or clear from the context, we sometimes also say allocation rather than $\mathcal{I}$-allocation. In this paper, we consider the following isolation levels: read committed (RC), snapshot isolation (SI), and serializable snapshot isolation (SSI). In general, with the exception of Section 5, $\mathcal{I} = \{\text{RC}, \text{SI}, \text{SSI}\}$. Before we define what it means for a schedule to consist of transactions adhering to different isolation levels, we introduce some necessary terminology.

Let s be a schedule for a set $\mathcal{T}$ of transactions. Two transactions $T_i, T_j \in \mathcal{T}$ are said to be *concurrent* in s when their execution overlaps. That is, if $\mathit{first}(T_i) <_s \mathtt{C}_j$ and $\mathit{first}(T_j) <_s \mathtt{C}_i$. In our running example schedule s_1 in Figure 2, T_1 and T_2 are concurrent, as well as T_1 and T_3. All other pairs of transactions are not concurrent in s_1. We say that a write operation $\mathtt{W}_j[\mathtt{t}]$ in a transaction $T_j \in \mathcal{T}$ *respects the commit order of s* if the version of $\mathtt{t}$ written by T_j is installed after all versions of $\mathtt{t}$ installed by transactions committing before T_j commits, but before all versions of $\mathtt{t}$ installed by transactions committing after T_j commits. More formally, if for every write operation $\mathtt{W}_i[\mathtt{t}]$ in a transaction $T_i \in \mathcal{T}$ different from T_j we have $\mathtt{W}_j[\mathtt{t}] \ll_s \mathtt{W}_i[\mathtt{t}]$ iff $\mathtt{C}_j <_s \mathtt{C}_i$. All write operations in Figure 2 respect the commit order of s_1. In Figure 4, write operations $\mathtt{W}_3[\mathtt{q}]$ and $\mathtt{W}_2[\mathtt{q}]$ do not respect the commit order of s_2 as $\mathtt{C}_2 <_{s_2} \mathtt{C}_3$ but $\mathtt{W}_3[\mathtt{q}] \ll_{s_2} \mathtt{W}_2[\mathtt{q}]$.

We next define when a read operation $a \in T$ reads the last committed version relative to a specific operation. For RC this operation is a itself while for SI this operation is $\mathit{first}(T)$. Intuitively, these definitions enforce that read operations in transactions allowed under RC act as if they observe a snapshot taken right before the read operation itself, while under SI they observe a snapshot taken right before the first operation of the transaction. A read operation $\mathtt{R}_j[\mathtt{t}]$ in a transaction $T_j \in \mathcal{T}$ is *read-last-committed in s relative to an operation $a_j \in T_j$* (not necessarily different from $\mathtt{R}_j[\mathtt{t}]$) if the following holds:

- $v_s(\mathtt{R}_j[\mathtt{t}]) = op_0$ or $\mathtt{C}_i <_s a_j$ with $v_s(\mathtt{R}_j[\mathtt{t}]) \in T_i$; and
- there is no write operation $\mathtt{W}_k[\mathtt{t}] \in T_k$ with $\mathtt{C}_k <_s a_j$ and $v_s(\mathtt{R}_j[\mathtt{t}]) \ll_s \mathtt{W}_k[\mathtt{t}]$.

The first condition says that $\mathtt{R}_j[\mathtt{t}]$ either reads the initial version or a committed version, while the second condition states that $\mathtt{R}_j[\mathtt{t}]$ observes the most recently committed version of $\mathtt{t}$ (according to $\ll_s$). For example, $\mathtt{R}_1[\mathtt{v}]$ in Figure 2 is read-last-committed in s_1 relative to $\mathtt{R}_1[\mathtt{v}]$ but not

to *first*(T_1), whereas in Figure 4 read operation $\mathtt{R}_3[\mathtt{v}]$ is read-last-committed in s_2 relative to *first*(T_3) but not relative to $\mathtt{R}_3[\mathtt{v}]$.

A transaction $T_j \in \mathcal{T}$ *exhibits a concurrent write in* s if there is another transaction $T_i \in \mathcal{T}$ and there are two write operations b_i and a_j in s on the same object with $b_i \in T_i$, $a_j \in T_j$ and $T_i \neq T_j$ such that $b_i <_s a_j$ and *first*(T_j) $<_s \mathtt{C}_i$. That is, transaction T_j writes to an object that has been modified earlier by a concurrent transaction T_i. A transaction $T_j \in \mathcal{T}$ *exhibits a dirty write in* s if there are two write operations b_i and a_j in s with $b_i \in T_i$, $a_j \in T_j$ and $T_i \neq T_j$ such that $b_i <_s a_j <_s \mathtt{C}_i$. That is, transaction T_j writes to an object that has been modified earlier by T_i, but T_i has not yet issued a commit. Notice that by definition a transaction exhibiting a dirty write always exhibits a concurrent write. Transaction T_1 in Figure 2 exhibits a concurrent write in s_1, since it writes to $\mathtt{v}$, which has been modified earlier by a concurrent transaction T_3. However, T_1 does not exhibit a dirty write in s_1, since T_3 has already committed before T_1 writes to $\mathtt{v}$. In Figure 4, T_2 exhibits a dirty write in s_2 since it writes to $\mathtt{q}$, which has been modified earlier by T_3 and T_3 has not yet committed before $\mathtt{W}_2[\mathtt{q}]$.

Definition 2.3. *Let s be a schedule over a set of transactions $\mathcal{T}$. A transaction $T_i \in \mathcal{T}$ is* allowed under isolation level read committed (RC) in s *if:*

- *write operations in T_i respect the commit order of s;*
- *each read operation $b_i \in T_i$ is read-last-committed in s relative to b_i; and*
- *T_i does not exhibit dirty writes in s.*

A transaction $T_i \in \mathcal{T}$ is allowed under isolation level snapshot isolation (SI) in s *if:*

- *write operations in T_i respect the commit order of s;*
- *each read operation in T_i is read-last-committed in s relative to first(T_i); and*
- *T_i does not exhibit concurrent writes in s.*

We then say that the schedule s is allowed under RC (respectively, SI) if every transaction is allowed under RC (respectively, SI) in s. The latter definitions correspond to the ones in the literature (see, e.g., [19, 25]). We emphasize that our definition of RC is based on concrete implementations over multiversion databases, found in e.g. Postgres, and should therefore not be confused with different interpretations of the term Read Committed, such as lock-based implementations [12] or more abstract specifications covering a wider range of concrete implementations (see, e.g., [2]). In particular, abstract specifications such as [2] do not require the read-last-committed property, thereby facilitating implementations in distributed settings, where read operations are allowed to observe outdated versions. When studying robustness, such a broad specification of RC is not desirable, since it allows for a wide range of schedules that are not conflict-serializable. We furthermore point out that our definitions of RC and SI are not strictly weaker forms of conflict-serializability. That is, a conflict-serializable schedule is not necessarily allowed under RC and SI as well.

While RC and SI are defined on the granularity of a single transaction, SSI enforces a global condition on the schedule as a whole. For this, recall the concept of dangerous structures from [14]: three transactions $T_1, T_2, T_3 \in \mathcal{T}$ (where T_1 and T_3 are not necessarily different) form a *dangerous structure* $T_1 \to T_2 \to T_3$ in s if:

- there is a rw-antidependency from T_1 to T_2 and from T_2 to T_3 in s;
- T_1 and T_2 are concurrent in s;
- T_2 and T_3 are concurrent in s;
- $\mathtt{C}_3 \leq_s \mathtt{C}_1$ and $\mathtt{C}_3 <_s \mathtt{C}_2$; and
- if T_1 is read-only, then $\mathtt{C}_3 <_s$ *first*(T_1).

Note that this definition of dangerous structures slightly extends upon the one in [14], where it is not required for T_3 to commit before T_1 and T_2. In the full version [15] of that paper, it is shown that such a structure can only lead to non-serializable schedules if T_3 commits first, and actual implementations of SSI (e.g., Postgres [23]) therefore include this optimization when monitoring for dangerous structures to reduce the number of aborts due to false positives.

We are now ready to define when a schedule is allowed under a (mixed) allocation of isolation levels.

Definition 2.4. *A schedule s over a set of transactions $\mathcal{T}$* is allowed under an allocation $\mathcal{A}$ *over $\mathcal{T}$ if:*

- *for every transaction $T_i \in \mathcal{T}$ with $\mathcal{A}(T_i) = RC$, T_i is allowed under RC;*
- *for every transaction $T_i \in \mathcal{T}$ with $\mathcal{A}(T_i) \in \{SI, SSI\}$, T_i is allowed under SI; and*
- *there is no dangerous structure $T_i \to T_j \to T_k$ in s formed by three (not necessarily different) transactions $T_i, T_j, T_k \in \{T \in \mathcal{T} \mid \mathcal{A}(T) = SSI\}$.*

We denote the allocation mapping all transactions to RC (respectively, SI) by $\mathcal{A}_{\text{RC}}$ (respectively, $\mathcal{A}_{\text{SI}}$).

In our running example schedule s_1 in Figure 2, the first three transactions form a dangerous structure $T_3 \to T_1 \to T_2$, witnessed by the rw-antidependencies $\mathtt{R}_3[\mathtt{v}] \to_{s_1} \mathtt{W}_1[\mathtt{v}]$ and $\mathtt{R}_1[\mathtt{t}] \to_{s_1} \mathtt{W}_2[\mathtt{t}]$. Therefore, s_1 is not allowed under an allocation mapping all three transactions to SSI. Since $\mathtt{R}_1[\mathtt{v}]$ is read-last-committed relative to itself but not relative to the first operation of T_1, and since T_1 exhibits a concurrent write, T_1 is allowed under RC in s_1, but not allowed under SI. Transactions T_2, T_3 and T_4 are allowed under both RC and SI in s_1. Notice in particular that all read operations in T_3 and T_4 are read-last-committed relative to both themselves and the first operation of the corresponding transaction in s_1. We conclude that s_1 is allowed under all allocations over $\mathcal{T}_{\text{ex}}$ mapping T_1 to RC, and T_2, T_3 and T_4 to either RC, SI or SSI. For schedule s_2 in Figure 4, no allocation $\mathcal{A}$ over $\mathcal{T}_{\text{ex}}$ exists such that s_2 is allowed under $\mathcal{A}$. Indeed, not all write operations in T_2 and T_3 respect the commit order of s_2, and T_2 furthermore exhibits a dirty write, which is not allowed under any isolation level.

2.4 Robustness

We define the robustness property [13] (also called *acceptability* in [19, 20]), which guarantees serializability for all schedules over a given set of transactions for a given allocation.

	T_1	T_2	T_3	T_4	$\mathcal{T}_{\text{ex}}$ robust against $\mathcal{A}_i$?
$\mathcal{A}_1$	RC	RC	SSI	SSI	no (cf. s_1 in Figure 2)
$\mathcal{A}_2$	SSI	RC	SSI	SSI	yes
$\mathcal{A}_3$	SI	SI	SSI	SSI	yes
$\mathcal{A}_4$	SI	RC	SSI	SSI	yes
$\mathcal{A}_5$	SI	RC	SI	SSI	no (cf. s_3 in Figure 6)
$\mathcal{A}_6$	SI	RC	SSI	SI	no (cf. s_3 in Figure 6)

Table 1: Example allocations for $\mathcal{T}_{\text{ex}}$.

Definition 2.5 (Robustness). *A set of transactions $\mathcal{T}$ is* robust *against an allocation $\mathcal{A}$ for $\mathcal{T}$ if every schedule for $\mathcal{T}$ that is allowed under $\mathcal{A}$ is conflict-serializable.*

We refer to $\mathcal{A}$ as a *robust allocation*. The *robustness problem* is then to decide whether a given allocation for a set of transactions $\mathcal{T}$ is a robust allocation.

Table 1 presents some allocations over our running example $\mathcal{T}_{\text{ex}}$. For each allocation that $\mathcal{T}_{\text{ex}}$ is not robust against, a counterexample schedule is given. For example, $\mathcal{T}$ is not robust against $\mathcal{A}_1$ because schedule s_1 in Figure 2 is allowed under $\mathcal{A}_1$ and not conflict-serializable.

3. DECIDING ROBUSTNESS

In the next definition, for an operation $b \in T$, we denote by $\mathsf{prefix}_b(T)$ the restriction of T to all operations that are before or equal to b according to $\leq_T$. Similarly, we denote by $\mathsf{postfix}_b(T)$ the restriction of T to all operations that are strictly after b according to $\leq_T$.

Definition 3.1 (Multiversion split schedule). *Let $\mathcal{T} = \{T_1, T_2, \ldots, T_n\}$ be a set of transactions and $\mathcal{A}$ an allocation for $\mathcal{T}$. A* multiversion split schedule *s for $\mathcal{T}$ and $\mathcal{A}$ is a multiversion schedule allowed under $\mathcal{A}$ that has the following form:*

$$\mathsf{prefix}_{b_1}(T_1) \cdot T_2 \cdot \ldots \cdot T_m \cdot \mathsf{postfix}_{b_1}(T_1) \cdot T_{m+1} \cdot \ldots \cdot T_n,$$

where $b_1 \in T_1$ and $m \in [2, n]$. Furthermore, for each pair of transactions $T_i, T_j \in \mathcal{T}$ with $i, j \in [1, m]$ and either $j = i+1$ or $i = m$ and $j = 1$, there are two operations $b_i \in T_i$ and $a_j \in T_j$ such that $b_i \rightarrow_s a_j$.

Schedule s_1 in Figure 3 is a multiversion split schedule for our running example set of transactions $\mathcal{T}_{\text{ex}}$ and allocation $\mathcal{A}_1$. Notice in particular the cyclic chain of dependencies implied by the definition. Because of this, such a schedule is never conflict-serializable. The existence of a multiversion split schedule for a set of transactions $\mathcal{T}$ and an allocation $\mathcal{A}$ for $\mathcal{T}$ therefore implies that $\mathcal{T}$ is not robust against $\mathcal{A}$. The next Theorem states that the opposite direction is also true, thereby characterizing non-robustness in terms of the existence of a multiversion split schedule.

Theorem 3.2. *For a set of transactions $\mathcal{T}$ and an allocation $\mathcal{A}$ for $\mathcal{T}$, the following are equivalent:*

1. *$\mathcal{T}$ is not robust against $\mathcal{A}$;*
2. *there is a multiversion split schedule s for $\mathcal{T}$ and $\mathcal{A}$.*

Theorem 3.2 forms the basis for a PTIME algorithm deciding robustness against a given allocation, presented as Algorithm 1 in [27]. We emphasize that a naive approach enumerating all multiversion split schedules for $\mathcal{T}$ and $\mathcal{A}$ does not work, as this number can still be exponential in the number of transactions in $\mathcal{T}$. Instead, the algorithm relies on a number of conditions on $\mathcal{A}$ as well as the the operations in $\mathcal{T}$ and makes use of an auxiliary graph structure to efficiently check whether a multiversion split schedule exists that witnesses the desired cyclic chain of dependencies and is allowed under $\mathcal{A}$. We refer to [27] for the details of the algorithm.

Theorem 3.3 ([27]). *Algorithm 1 in [27] decides whether a set of transactions $\mathcal{T}$ is robust against an allocation $\mathcal{A}$ in time $O(|\mathcal{T}|^3 \cdot max\{|\mathcal{T}|^3, k^2\ell^2, \ell^6\})$, with k the total number of operations in $\mathcal{T}$ and ℓ the maximum number of operations in a transaction in $\mathcal{T}$.*

4. THE ALLOCATION PROBLEM

Finding a robust allocation over $\{\text{RC}, \text{SI}, \text{SSI}\}$ is of course trivial as we can simply assign every transaction to SSI. Such an allocation is undesirable as it enforces the most expensive concurrency control mechanism on all transactions. We are therefore interested in robust allocations that favor RC over SI and SI over SSI.

In the following, we assume a total order[2] RC $<$ SI $<$ SSI over the isolation levels, and introduce the following notions. Let $\mathcal{T}$ be a set of transactions, and let $\mathcal{A}$ and $\mathcal{A}'$ be allocations over $\mathcal{T}$. We denote by $\mathcal{A} \leq \mathcal{A}'$ when $\mathcal{A}(T) \leq \mathcal{A}'(T)$ for all $T \in \mathcal{T}$. Furthermore, $\mathcal{A} < \mathcal{A}'$ when $\mathcal{A} \leq \mathcal{A}'$ and there is a $T \in \mathcal{T}$ with $\mathcal{A}(T) < \mathcal{A}'(T)$.

We say that a robust allocation $\mathcal{A}$ is *optimal* when there is no robust allocation $\mathcal{A}'$ with $\mathcal{A}' < \mathcal{A}$. For an isolation level I, we denote by $\mathcal{A}[T \mapsto I]$ the allocation where T is assigned I and every other transaction $T' \in \mathcal{T}$ is assigned $\mathcal{A}(T')$. For two isolation levels $\mathcal{I}$ and $\mathcal{I}'$ with $\mathcal{I} < \mathcal{I}'$ (respectively $\mathcal{I}' < \mathcal{I}$) we say that $\mathcal{I}$ is a lower (respectively higher) isolation level than $\mathcal{I}'$.

The following proposition obtains some useful properties of robust allocations. Specifically, it says that robustness propagates upwards. That is, if a schedule is robust under an allocation $\mathcal{A}$, it remains robust when assigning a higher isolation level to any of its transactions T. Furthermore, if there exists a robust allocation $\mathcal{A}'$ mapping T to a lower isolation level than $\mathcal{A}(T)$, then $\mathcal{A}(T)$ can be safely updated to that lower isolation as well. That is, s is also robust under $\mathcal{A}[T \mapsto \mathcal{A}'(T)]$.

Proposition 4.1. *Let $\mathcal{T}$ be a set of transactions. Let $\mathcal{A}$ and $\mathcal{A}'$ be allocations for $\mathcal{T}$.*

1. *If $\mathcal{A} \leq \mathcal{A}'$ and $\mathcal{T}$ is robust against $\mathcal{A}$, then $\mathcal{T}$ is robust against $\mathcal{A}'$.*
2. *If $\mathcal{T}$ is robust against $\mathcal{A}$ and $\mathcal{A}'$, then $\mathcal{T}$ is robust against $\mathcal{A}'[T \mapsto \mathcal{A}(T)]$ for every $T \in \mathcal{T}$.*

Applying these results on the allocations in Table 1, $\mathcal{A}_4$ being a robust allocation implies that $\mathcal{A}_2$ and $\mathcal{A}_3$ are robust allocations as well. Furthermore, the robust allocations $\mathcal{A}_2$ and $\mathcal{A}_3$ can be combined into the robust allocation $\mathcal{A}_4$.

We can now prove the following proposition.

Proposition 4.2. *There is a unique optimal allocation for every set of transactions $\mathcal{T}$.*

[2]This order only represents the preference between isolation levels (i.e., RC over SI and SI over SSI), *not* an inclusion relation between isolation levels. For example, not every schedule allowed under $\mathcal{A}_{\text{SI}}$ is allowed under $\mathcal{A}_{\text{RC}}$ (cf. Example 5.2).

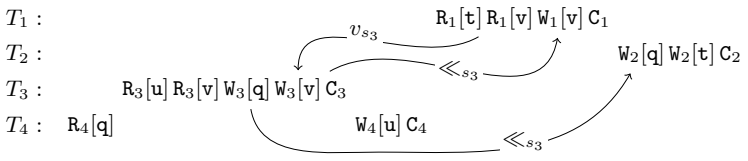


Figure 6: A schedule s_3 for $\mathcal{T}_{\text{ex}}$ with v_{s_3} and $\ll_{s_3}$ represented through arrows. The special operation op_0 and all arrows involving op_0 are omitted.

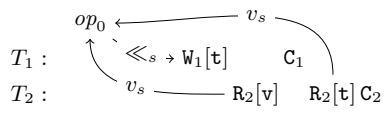


Figure 7: Schematic representation of schedule s in Example 5.2.

Algorithm 1: Computing the optimal robust allocation.

Input : Set of transactions $\mathcal{T}$
Output: Optimal robust allocation $\mathcal{A}$ for $\mathcal{T}$

$\mathcal{A} := \mathcal{A}_{\text{SSI}}$;
for $T \in \mathcal{T}$ **do**
  if $\mathcal{T}$ *is robust against* $\mathcal{A}[T \mapsto RC]$ **then**
    $\mathcal{A} := \mathcal{A}[T \mapsto \text{RC}]$;
  else if $\mathcal{T}$ *is robust against* $\mathcal{A}[T \mapsto SI]$ **then**
    $\mathcal{A} := \mathcal{A}[T \mapsto \text{SI}]$;
return $\mathcal{A}$

For our running example $\mathcal{T}_{\text{ex}}$, the optimal robust allocation is $\mathcal{A}_4$ in Table 1. Indeed, trying to lower the isolation level for one of the transactions even further always leads to nonrobust allocations (cf., $\mathcal{A}_1$, $\mathcal{A}_5$ and $\mathcal{A}_6$)

The following theorem shows that the unique optimal allocation can be computed in polynomial time. The corresponding algorithm is given as Algorithm 1.

THEOREM 4.3. *An optimal robust allocation can be computed in time polynomial in the size of $\mathcal{T}$ for every set of transactions $\mathcal{T}$.*

5. RESTRICTING TO RC AND SI

As already mentioned in the introduction, Oracle restricts to the isolation levels RC and SI. We investigate in this section how the results of the previous sections can be transferred to this setting. In particular, we ignore SSI and restrict attention to RC and SI.

We start with the following result.

PROPOSITION 5.1. *For a set of transactions $\mathcal{T}$, robustness against $\mathcal{A}_{RC}$ implies robustness against $\mathcal{A}_{SI}$.*

This above result is an immediate consequence of Proposition 5.4. We mention that it is also a direct consequence of the characterizations for robustness against $\mathcal{A}_{\text{RC}}$ [25] and $\mathcal{A}_{\text{SI}}$ [19]. Indeed, it can be shown that a counterexample for robustness against $\mathcal{A}_{\text{SI}}$ can always be transformed into a counterexample for robustness against $\mathcal{A}_{\text{RC}}$ as well. We do want to emphasize that Proposition 5.1 is *not* a trivial consequence that immediately follows from the definitions of the isolation levels RC and SI, for the simple reason that it is not the case that every schedule allowed under $\mathcal{A}_{\text{SI}}$ is also allowed under $\mathcal{A}_{\text{RC}}$ as the next example shows.

EXAMPLE 5.2. *We give an example of a schedule s that is allowed under SI but not allowed under RC. To this end, consider the schedule s over transactions $\mathtt{W}_1[\mathtt{t}]\,\mathtt{C}_1$ and $\mathtt{R}_2[\mathtt{v}]\,\mathtt{R}_2[\mathtt{t}]\,\mathtt{C}_2$ with operation order $\leq_s$,*

$$op_0\,\mathtt{W}_1[\mathtt{t}]\,\mathtt{R}_2[\mathtt{v}]\,\mathtt{C}_1\,\mathtt{R}_2[\mathtt{t}]\,\mathtt{C}_2,$$

version order $op_0 \ll_s \mathtt{W}_1[\mathtt{t}]$, and version function $v_s(\mathtt{R}_2[\mathtt{v}]) = v_s(\mathtt{R}_2[\mathtt{t}]) = op_0$. Figure 7 shows a graphical representation of schedule s. Then, s is allowed under $\mathcal{A}_{SI}$, but not under $\mathcal{A}_{RC}$, because $\mathtt{R}_2[\mathtt{t}]$ is not read-last-committed in s relative to itself. □

We formalize when a set of transactions is robustly allocatable against a class of isolation levels:

DEFINITION 5.3. *For a class of isolation levels $\mathcal{I}$, a set of transactions $\mathcal{T}$ is* robustly allocatable against $\mathcal{I}$ *if there exists an $\mathcal{I}$-allocation $\mathcal{A}$ such that $\mathcal{T}$ is robust against $\mathcal{A}$.*

The only if-direction of the next theorem now immediately follows from Proposition 4.1(1) as $\mathcal{A} \leq \mathcal{A}_{\text{SI}}$ for any $\{\text{RC}, \text{SI}\}$-allocation $\mathcal{A}$ for which a set of transactions is robustly allocatable. The if-direction is trivial, since robustness against $\mathcal{A}_{\text{SI}}$ is an immediate witness for $\mathcal{T}$ being robustly allocatable against $\{\text{RC}, \text{SI}\}$:

PROPOSITION 5.4. *A set of transactions $\mathcal{T}$ is robustly allocatable against $\{RC, SI\}$ iff $\mathcal{T}$ is robust against $\mathcal{A}_{SI}$.*

We now state the main result of this section:

THEOREM 5.5. *Let $\mathcal{T}$ be a set of transactions. It can be decided in time polynomial in the size of $\mathcal{T}$ whether $\mathcal{T}$ is robustly allocatable against $\{RC, SI\}$. If $\mathcal{T}$ is robustly allocatable against $\{RC, SI\}$, then an optimal unique allocation can be computed in polymonial time as well.*

6. RELATED WORK

Mixing isolation levels. Adya et al. [2] define isolation levels in terms of phenomena that are forbidden to occur in the serialization graph. they consider a mixture of READ UNCOMMITTED, READ COMMITTED and serializable transactions and do not consider SI or SSI like we do in this paper. Note that a separate graph-based definition of SI is specified in [1], requiring an extension of the serialization graph. Incorporating SI in the mixed setting in [2] is therefore not trivial. Other work [13, 19] consider a limited form of isolation level mixing where one isolation level (say, SI) can be mixed with a serializable isolation level. *To the best of our knowledge, this paper is the first that jointly considers mixing RC, SI and SSI in the way that it is applied in Postgres.*

Robustness and allocation for transactions. Fekete [19] is the first work that provides a necessary and sufficient condition for deciding robustness against an isolation level (SI) for a workload of transactions. In particular, that work pro-

vides a characterization for optimal allocations when every transaction runs under either SI or strict two-phase locking (S2PL). As a side result, this work presents a characterization for robustness against SI as well.

Ketsman et al. [22] provide characterisations for robustness against READ COMMITTED and READ UNCOMMITTED under lock-based semantics. In addition, it is shown that the corresponding decision problems are complete for coNP and LOGSPACE, respectively, which should be contrasted with the polynomial time characterization obtained in [25] for robustness against *multiversion* read committed which is the variant that is considered in this paper. *The present paper is therefore the first to address the robustness and allocation problem for a wider range of isolation levels.*

Robustness in practice. The setting in the present paper assumes that the complete set of all transactions in a workload is completely known which is an assumption that can not always be met in practice. In [27] we provide an elaborate discussion of different approaches that have been previously investigated to address this [6, 8, 13, 16–18, 20, 21, 25, 26].

7. CONCLUSION

In this paper, we addressed and solved the robustness and allocation problem for the classes {RC, SI, SSI} and {RC, SI} corresponding to the isolation levels employed in Postgres and Oracle, respectively. These results can be used as a stepping stone for corresponding results on the level of transaction programs, thereby laying the groundwork for automating isolation level allocation within existing databases that support multiversion concurrency control.

Acknowledgments

This work is partly funded by FWO-grant G019921N. We thank Alan Fekete for spotting the omission related to read-only transactions in our definition of dangerous structures.

8. REFERENCES

[1] A. Adya. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions*. Ph.D., MIT, Cambridge, MA, USA, Mar. 1999.

[2] A. Adya, B. Liskov, and P. E. O'Neil. Generalized isolation level definitions. In *ICDE*, pages 67–78, 2000.

[3] M. Alomari. Serializable executions with snapshot isolation and two-phase locking: Revisited. In *AICCSA*, pages 1–8, 2013.

[4] M. Alomari, M. Cahill, A. Fekete, and U. Rohm. The cost of serializability on platforms that use snapshot isolation. In *ICDE*, pages 576–585, 2008.

[5] M. Alomari, M. J. Cahill, A. D. Fekete, and U. Röhm. Serializable executions with snapshot isolation: Modifying application code or mixing isolation levels? In *DASFAA*, volume 4947, pages 267–281, 2008.

[6] M. Alomari and A. Fekete. Serializable use of read committed isolation level. In *AICCSA*, pages 1–8, 2015.

[7] M. Alomari, A. D. Fekete, and U. Röhm. A robust technique to ensure serializable executions with snapshot isolation DBMS. In *ICDE*, pages 341–352, 2009.

[8] P. Alvaro and K. Kingsbury. Elle: Inferring isolation anomalies from experimental observations. *PVLDB*, 14(3):268–280, 2020.

[9] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Highly available transactions: Virtues and limitations. *PVLDB*, 7(3):181–192, 2013.

[10] S. M. Beillahi, A. Bouajjani, and C. Enea. Checking robustness against snapshot isolation. In *CAV*, pages 286–304, 2019.

[11] S. M. Beillahi, A. Bouajjani, and C. Enea. Robustness against transactional causal consistency. In *CONCUR*, pages 1–18, 2019.

[12] H. Berenson, P. A. Bernstein, J. Gray, J. Melton, E. J. O'Neil, and P. E. O'Neil. A critique of ANSI SQL isolation levels. In *SIGMOD*, pages 1–10, 1995.

[13] G. Bernardi and A. Gotsman. Robustness against consistency models with atomic visibility. In *CONCUR*, pages 7:1–7:15, 2016.

[14] M. J. Cahill, U. Röhm, and A. D. Fekete. Serializable isolation for snapshot databases. In *SIGMOD*, pages 729–738, 2008.

[15] M. J. Cahill, U. Röhm, and A. D. Fekete. Serializable isolation for snapshot databases. *ACM Trans. Database Syst.*, 34(4):20:1–20:42, 2009.

[16] A. Cerone, G. Bernardi, and A. Gotsman. A framework for transactional consistency models with atomic visibility. In *CONCUR*, pages 58–71, 2015.

[17] A. Cerone and A. Gotsman. Analysing snapshot isolation. *J.ACM*, 65(2):1–41, 2018.

[18] A. Cerone, A. Gotsman, and H. Yang. Algebraic Laws for Weak Consistency. In *CONCUR*, pages 26:1–26:18, 2017.

[19] A. Fekete. Allocating isolation levels to transactions. In *PODS*, pages 206–215, 2005.

[20] A. Fekete, D. Liarokapis, E. J. O'Neil, P. E. O'Neil, and D. E. Shasha. Making snapshot isolation serializable. *ACM Trans. Database Syst.*, 30(2):492–528, 2005.

[21] Y. Gan, X. Ren, D. Ripberger, S. Blanas, and Y. Wang. Isodiff: Debugging anomalies caused by weak isolation. *PVLDB*, 13(11):2773–2786, 2020.

[22] B. Ketsman, C. Koch, F. Neven, and B. Vandevoort. Deciding robustness for lower SQL isolation levels. In *PODS*, pages 315–330, 2020.

[23] D. R. K. Ports and K. Grittner. Serializable snapshot isolation in postgresql. *PVLDB*, 5(12):1850–1861, 2012.

[24] TPC-C. On-line transaction processing benchmark. `http://www.tpc.org/tpcc/`.

[25] B. Vandevoort, B. Ketsman, C. Koch, and F. Neven. Robustness against read committed for transaction templates. *PVLDB*, 14(11):2141–2153, 2021.

[26] B. Vandevoort, B. Ketsman, C. Koch, and F. Neven. Robustness against read committed for transaction templates with functional constraints. In *ICDT*, pages 16:1–16:17, 2022.

[27] B. Vandevoort, B. Ketsman, and F. Neven. Allocating isolation levels to transactions in a multiversion setting. In *PODS*, pages 69–78, 2023.

Technical Perspective: From Binary Join to Free Join

Thomas Neumann
TUM
neumann@in.tum.de

Most queries access data from more than one relation, which makes joins between relations an extremely common operation. In many cases the execution time of a query is dominated by the processing of the involved joins. This observation has led to a wide range of techniques to speed up join processing like, e.g. efficient hash joins, bitmap filters to eliminate non-joining tuples early on, blocked lookups to hide cache latencies, and many others.

But even the most careful engineering technique cannot hide the fact that a join is fundamentally a growing operation, at least in the general case. This problem is not immediately visible because many real-world joins are foreign key / primary key joins, which means that we will have at most one join partner per foreign key, and thus the result of the join will not be larger than the inputs. But that is not always the case. In general, a join is a n:m combination, which means that the result of a join can be as large as $O(n^2)$, where n is the input size. When n is large, as it is often case for databases, this can lead to disastrous execution times.

An interesting observation in this context is that a $O(n^2)$ (intermediate) result is usually a mistake. Very few users will ask queries where the result consists of millions or billions of tuples. Usually there will be subsequent operations that eliminate most of these intermediate results until only the query result remains. The query optimizer will try to find a join execution order where the intermediate result sizes are minimized, but in the presence of growing joins and often unreliable cardinality estimates that is not an easy task. Sometimes small changes the query lead to very different execution times, which is unfortunate for the robustness of the system.

And fundamentally, some queries with growing joins are inherently hard for binary joins, independent of join order. The classic example for that are triangle queries, for example the natural join between the sets $R(a,b), S(b,c), T(c,a)$. These queries often have very large intermediate result but only small query results. And even worse, it can be shown that for some data sets any strategy that uses binary joins will have $O(n^2)$ runtime, which makes these queries intractable for large data sets. While the query result itself is only in $O(n^{1.5})$ even in worst case [1], and in reality it will usually be much smaller.

Another class of join algorithms, so called worst-case optimal joins (WCOJs), process joins not as a sequence of binary joins but as a sequence of attribute equivalence classes, where all relations involved in the equivalence class are processed within the step. This avoids the intermediate result explosion and makes triangle queries tractable. But they are rarely used in database systems because the constant factors in the implementation are higher and they tend to be slower than traditional binary queries for the more common case that the intermediate results do not grow dramatically [2]. But simply always using optimized binary joins is dangerous, they are faster on typical queries but can exhibit catastrophic performance for growing queries.

The next paper finds an interesting compromise between these two extremes. It introduces so called *free join* plans, which can process both multiple attributes and multiple relations in each step. This makes both traditional binary joins and the *general join* WCOJ algorithm special cases of free joins, as both can be expressed in the framework. In addition, new strategies can be expressed as free join plans that combine aspects of traditional joins and WCOJs in one execution plan.

This allows for avoiding the high constant factors of WCOJs for the parts of the query where they are not required, while still retaining the superior pruning power of WCOJs in the overall query. This is a very nice result that will hopefully lead to systems that are more robust and can process join queries predictable and reliable.

1. REFERENCES

[1] A. Atserias, M. Grohe, and D. Marx. Size bounds and query plans for relational joins. In *49th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2008, October 25-28, 2008, Philadelphia, PA, USA*, pages 739–748. IEEE Computer Society, 2008.

[2] M. J. Freitag, M. Bandle, T. Schmidt, A. Kemper, and T. Neumann. Adopting worst-case optimal joins in relational database systems. *Proc. VLDB Endow.*, 13(11):1891–1904, 2020.

From Binary Join to Free Join

Yisu Remy Wang
UCLA
Los Angeles, CA, USA

Max Willsey
UC Berkeley
Berkeley, CA, USA

Dan Suciu
University of Washington
Seattle, WA, USA

ABSTRACT

Over the last decade, worst-case optimal join (WCOJ) algorithms have emerged as a new paradigm for one of the most fundamental challenges in query processing: computing joins efficiently. Such an algorithm can be asymptotically faster than traditional binary joins, all the while remaining simple to understand and implement. However, they have been found to be less efficient than the old paradigm, traditional binary join plans, on the typical acyclic queries found in practice. In an effort to unify and generalize the two paradigms, we proposed a new framework, called Free Join, in our SIGMOD 2023 paper. Not only does Free Join unite the worlds of traditional and worst-case optimal join algorithms, it uncovers optimizations and evaluation strategies that outperform both.

In this article, we approach Free Join from the traditional perspective of binary joins, and re-derive the more general framework via a series of gradual transformations. We hope this perspective from the past can help practitioners better understand the Free Join framework, and find ways to incorporate some of the ideas into their own systems.

1. INTRODUCTION

Over the last decade, worst-case optimal join (WCOJ) algorithms [10, 14, 11, 9] have emerged as a breakthrough in one of the most fundamental challenges in query processing: computing joins efficiently. Such an algorithm can be asymptotically faster than traditional binary joins, all the while remaining simple to understand and implement [11]. These algorithms opened up a flourishing field of research, leading to both theoretical results [11, 6] and practical implementations [14, 2, 4, 8].

Over time, a common belief took hold: "WCOJ is designed for cyclic queries". This belief is rooted in the observation that WCOJ enjoys lower asymptotic complexity than traditional algorithms for cyclic queries [11], but when the query is acyclic, classic algorithms like the Yannakakis algorithm [16] are already asymptotically optimal. Moreover, traditional binary join algorithms have benefited from decades of research and engineering. Techniques like column-oriented layout, vectorization, and query optimization have contributed compounding constant-factor speedups, making it challenging for WCOJ to be competitive in practice.

This is a minor revision of the paper entitled Free Join: Unifying Worst-Case Optimal and Traditional Joins, published in Proc. ACM Manag. Data, Vol. 1, No. 2, Article 150, June 2023.
2836-6573/2023/6-ART150 https://doi.org/10.1145/3589295

$$Q_{\clubsuit}(x,a,b,c) \text{ :- } R(x,a), S(x,b), T(x,c).$$

```
SELECT * FROM R, S, T WHERE R.x = S.x AND S.x = T.x
```

$$\begin{aligned} R =& \{(x_0,a_0)\} \cup \{(x_1,a_i^l),(x_2,a_i^r) \mid i \in [1\ldots n]\} \\ S =& \{(x_0,b_0)\} \cup \{(x_2,b_i^l),(x_3,b_i^r) \mid i \in [1\ldots n]\} \\ T =& \{(x_0,c_0)\} \cup \{(x_3,c_i^l),(x_1,c_i^r) \mid i \in [1\ldots n]\} \end{aligned}$$

Figure 1: The clover query $Q_{\clubsuit}$, and an input instance. Note that x_0 is the only x-value in all three relations, therefore the only output tuple is (x_0,a_0,b_0,c_0).

The dichotomy of WCOJ versus binary join has led researchers and practitioners to view the algorithms as opposites. In our SIGMOD 2023 paper [15], we broke down this dichotomy with a new framework called Free Join that unifies WCOJ and binary join. Further more, we proposed new data structures, evaluation algorithms, and optimizations to make Free Join outperform both binary join and WCOJ.

In this article, we review Free Join from a new perspective: starting from the traditional binary join algorithm, we apply a series of gradual transformations to arrive at the Free Join algorithm as well as the WCOJ algorithm. We hope this perspective from the past can help practitioners better understand Free Join, and pave the way for its adoption into existing systems.

This article is based on the paper Free Join: Unifying Worst-case Optimal and Traditional Joins [15], published at SIGMOD 2023.

2. FROM BINARY JOIN TO FREE JOIN

In this section we introduce the Free Join framework. Unlike our SIGMOD paper [15] which defines Free Join from the basic building blocks, here we start from the traditional binary join and gently massage it into the more general Free Join. To keep the presentation intuitive, we will be following an example instead of defining the algorithm in full generality. We refer the reader to our SIGMOD paper [15] for a more formal treatment.

2.1 Basic Concepts and Notations

For simplicity we consider only *natural join* queries, where all joins are equijoins, and all input relations are joined over common attributes. Such queries are also known as *conjunc-*

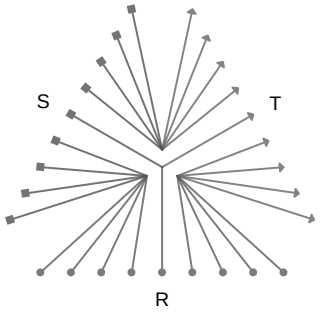


Figure 2: Visualization of the input relations to $Q_{\clubsuit}$. Each binary relation is represented as a set of edges on each side. The query $Q_{\clubsuit}$ looks for sets of 3 edges, one from each relation, that meet in the middle (there is only one such set, at the center of the figure).

```
for ...:
  m = M[x]?
  ...
```

```
for ...:
  if x not in M:
    continue
  else:
    m = M[x]
    ...
```

Figure 3: Example using the notation `m = M[x]?`. The two code fragments are equivalent.

tive queries and can be written in "Datalog notation" as the following example shows.

Example 1. *Consider SQL query in Figure 1. The corresponding conjunctive query appears above it, where each of $R(x,a)$, $S(x,b)$, and $T(x,c)$ is called a* body atom*, and $Q_{\clubsuit}(x,a,b,c)$ the* head atom.

It is often convenient to view a conjunctive query as a hypergraph. The *query hypergraph* of Q consists of vertices $\mathcal{V}$ and edges $\mathcal{E}$, where the set of nodes $\mathcal{V}$ is the set of variables occurring in Q, and the set of hyperedges $\mathcal{E}$ is the set of body atoms in Q. The hypergraph for $Q_{\clubsuit}$ has four vertices, each for x, a, b, and c, and three edges, each for $R(x,a)$, $S(x,b)$, and $T(x,c)$. As standard, we say that the query Q is *acyclic* if its associated hypergraph is α-acyclic[1] [3]. Note that $Q_{\clubsuit}$ is acyclic, while an example of a *cyclic* query is the "triangle query":

$$Q_{\triangle}(x,y,z) \text{ :- } U(x,y), V(y,z), W(z,x).$$

whose query hypergraph is a triangle.

We now introduce a notation to make pseudocode cleaner. Inside a loop, we will write `m = M[x]?` for looking up x from the hash map M; if M contains x, we assign the result of the lookup to m; otherwise, we `continue` to the next iteration of the enclosing loop. In other words, the code fragments in Figure 3 are equivalent.

2.2 Binary Join

The standard approach to computing a natural join of multiple relations is to compute one binary join at a time. A *binary plan* is a binary tree, where each internal node is a join operator $\bowtie$, and each leaf node is one of the base tables R_i. The plan is a *left-deep linear plan*, or simply left-deep plan, if the right child of every join is a leaf node. If the plan is not left-deep, then we call it *bushy*. For example, $(R \bowtie S) \bowtie (T \bowtie U)$ is a bushy plan, while $((R \bowtie S) \bowtie T) \bowtie U$ is a left-deep plan. We do not treat specially right-deep or zig-zag plans, but simply consider them to be bushy.

In this paper we consider only hash-joins, which are the most common types of joins in database systems. The standard way to execute a bushy plan is to decompose it into a series of left-deep linear plans. Every join node that is a right child becomes the root of a new subplan, which is first evaluated, and its result materialized, before the parent join can proceed. As a consequence, every binary plan, bushy or not, becomes a collection of left-deep plans. We decompose bushy plans in exactly the same way, and we will focus on left-deep linear plans in the rest of this paper. For example, the bushy plan $(R \bowtie S) \bowtie (T \bowtie U)$ is converted into two plans: $P_1 = T \bowtie U$ and $P_2 = (R \bowtie S) \bowtie P_1$; both are left-deep plans.

To reduce clutter, we represent a left-deep plan $(\cdots((R_1 \bowtie R_2) \bowtie R_3) \cdots \bowtie R_{m-1}) \bowtie R_m$ as $[R_1, R_2, \ldots, R_m]$. Evaluation of a left-deep plan is done using pipelining. The engine iterates over each tuple in the left-most base table R_1; each tuple is probed in R_2; each of the matching tuple is further probed in R_3, etc.

Example 2. *A possible left-deep linear plan for $Q_{\clubsuit}$ is $[R,S,T]$, which represents $(R(x,a) \bowtie S(x,b)) \bowtie T(x,c)$. To execute this plan, we first build a hash table for S keyed on x, where each x maps to a vector of (x,a) tuples, and a hash table for T keyed on x, each mapped to a vector of (x,c) tuples*[2]*. Then the execution proceeds as shown in Figure 4a. For each tuple (x,a) in R, we first probe into the hash table for S using x to get a vector of (x,b) tuples. We then loop over each (x,a) and probe into the hash table for T with x. Each successful probe will return a vector of (x,c) tuples, and we output the tuple (x,a,b,c) for each (x,c). On the input instance in Figure 1 (visualized in Figure 2), this algorithm runs in time $\Omega(n^2)$.*

2.3 Columnar Storage and Late Materialization

The first transformation we perform on the binary join algorithm makes it work on column-wise storage instead of a row-wise one. This is not yet an optimization because it likely will not improve the performance, but this step serves as an important bridge to the next optimizations. As Figure 4b shows, in the outermost loop we iterate over row indices instead of tuples. For each row index i, we retrieve the x-value $R.x[i]$, as well as the corresponding a-value $R.a[i]$. The hash maps for S and T now map each x to a vector of row indices, so we next look up into the hash map for S using x to get a vector of j. In the second loop, we retrieve the x-value and b-value from S for each j, then probe into T to get a vector of k. Finally, we retrieve the x-value and c-value from T for each k, and output the tuple (x,a,b,c).

A key inefficiency of the algorithm in Figure 4b is that, although the query only outputs a single tuple (x_0,a_0,b_0,c_0),

[1]The reader does not need to be familiar with definitions of acyclic queries to understand Free Join.

[2]When the relations are bags, then the hash table may contain duplicate tuples, or store separately the multiplicity. We also note that the question what exactly to store in the hash table (e.g. copies of the tuples, or pointers to the tuple in the buffer pool) has been studied for a long time, see [5].

```
for (x,a) in R:
  s = S[x]?
  for (x,b) in s:
    t = T[x]?
    for (x,c) in t:
      output(x,a,b,c)
```

(a) Binary Free Join.

```
for i in 0..R.len():
  x = R.x[i]; a = R.a[i]
  s = S[x]? # s:x->[j]
  for j in s:
    x = S.x[j]; b = S.b[j]
    t = T[x]? # t:x->[k]
    for k in t:
      x = T.x[k]; c = T.c[k]
      output(x,a,b,c)
```

(b) Columnar storage.

```
for i in 0..R.len():
  x = R.x[i]; s = S[x]?
  for j in s:
    x = S.x[j]; t = T[x]?
    for k in t:
      x = T.x[k]
      a = R.a[i]
      b = S.b[j]
      c = T.c[k]
      output(x,a,b,c)
```

(c) Late materialization.

```
for i in 0..R.len():
  x = R.x[i];
  s = S[x]?; t = T[x]?
  for j in s:
    for k in t:
      a = R.a[i]
      b = S.b[j]
      c = T.c[k]
      output(x,a,b,c)
```

(d) Late iteration.

Figure 4: Execution of binary join for the clover query 4a, and three transformations. The first transformation 4b makes the algorithm work on column-wise storage instead of a row-wise one; the second transformation 4c performs the classic late materialization optimization; the last one is another transformation that we call late iteration 4d.

we still did a lot of work retrieving the different a, b, and c values from their respective columns. For example, since we iterate over the entire R relation, we retrieve all $2n + 1$ a-values from $R.a$; even worse, since $|R \bowtie S| = n^2 + 1$, we will access $S.b$ $\Omega(n^2)$ times. A better strategy is to *delay* the retrieval of these values until we actually need them. In this case, we can delay the retrieval of a, b, and c until we are ready to output the tuple (x, a, b, c). This way we only need to access each of $R.a$, $S.b$, and $T.c$ once, instead of $\Omega(n^2)$ times. This is precisely the classic *late materialization* optimization [1] now implemented in nearly all modern database systems.

2.4 Late Iteration and Free Join

We can go one step beyond late materialization and further optimize the code in Figure 4c. The key observation is that, although we retrieve x from the different relations in each loop level, *they have to be the same value* because x is the join attribute! This means we can remove the last two redundant retrievals of x and reuse the value from the outermost loop, which corresponds to removing the underlined code in Figure 4c. At this point, we can see that the remaining body of the second loop, `t = T[x]?`, does not depend on the loop variable j at all. We can therefore pull the lookup out of the loop, resulting in the code in Figure 4d. In other words, we *delay* the iteration over s until after the lookup on T succeeds.

Note that this final optimization has improved the asymptotic run time of the algorithm: although late materialization already saves a quadratic number of accesses to the relation columns, it still needs to iterate over $\Omega(n^2)$ row indices, because the first two loop levels essentially compute the join $R \bowtie S$. In contrast, the first loop level in Figure 4d joins every tuple of R with S and T at the same time, and the entire algorithm now runs in $O(n)$ time.

At the moment, our optimizations may appear rather low-level and ad-hoc. Taking a step back, we can understand the execution of any join algorithm as a series of iterations and lookups. The transformation from row-wise to column-wise storage involves changing *what to iterate over* (row indices instead of tuples); the late materialization optimization changes *what to look up* and *when to look up* (look up row indices first, then retrieve values later); finally, the late iteration optimization reorders the iterations and lookups.

While columnar storage and late materialization have become stables of modern database systems, the contribution of the Free Join framework is a new abstraction to describe the ordering of iterations and lookups that we call the Free Join plan. The basic building blocks of a Free Join plan are called *subatoms*, each of which is a subset of a relation schema.

Definition 1. *Given a relation schema $R(x_1, x_2, \ldots)$, a* subatom *is of the form $R(x_i, x_j, \ldots)$ where $\{x_i, x_j, \ldots\} \subseteq \{x_1, x_2, \ldots\}$.*

For example, given the relation R with schema $R(x, y)$, all of the following are valid subatoms: $R(), R(x), R(y), R(x, y)$. A Free Join plan over a set of schemas is a sequence of groups, where each group is a list of subatoms.

Definition 2. *Given a join query Q over $R_1, R_2, \ldots$, a Free Join plan for Q is of the form:*

$$[R_i(\mathbf{x}_i), R_j(\mathbf{x_j}), \ldots], [R_k(\mathbf{x_k}), \ldots], \ldots$$

where each of $R_i(\mathbf{x_i}), R_j(\mathbf{x_j}), R_k(\mathbf{x_k}), \ldots$ is a subatom over the schema of $R_i, R_j, R_k, \ldots$ respectively.

Each group in a Free Join plan corresponds to a loop level. At each loop level, we iterate over tuples of the first subatom in the group, and use the values to look up into the remaining subatoms. For this reason, we will sometimes stylize a Free Join plan as follows to emphasize the iterated subatom and reflect the loop nesting:

$$\begin{aligned}&[R_1(\mathbf{x_1}) \mid R_2(\mathbf{x_2}), R_3(\mathbf{x_3}), \ldots]\\ &\rightarrow [R_4(\mathbf{x_4}) \mid R_5(\mathbf{x_2}), \ldots]\\ &\rightarrow [R_6(\mathbf{x_6}) \mid \ldots]\end{aligned}$$

Example 3. *The Free Join plan for the algorithm in Figure 4d is:*

$$\begin{aligned}&[R(x, a) \mid S(x), T(x)]\\ &\rightarrow [S(b) \mid]\\ &\rightarrow [T(c) \mid]\end{aligned}$$

Although we only retrieve the value of a in the innermost loop, each a-value one-to-one corresponds to each i, so the plan iterates over both x and a at the first level.

Example 4. *We can represent the binary join algorithm in Figure 4a with the Free Join plan:*

$$\begin{aligned}&[R(x, a) \mid S(x)]\\ &\rightarrow [S(b) \mid T(x)]\\ &\rightarrow [T(c) \mid]\end{aligned}$$

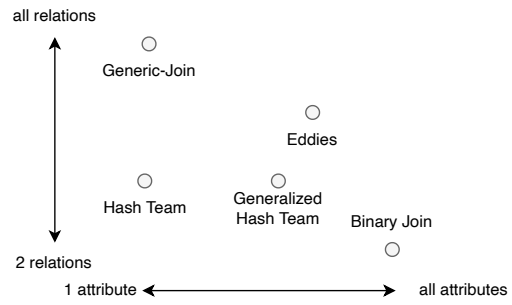


Figure 5: Free Join plans represent a design space of join algorithms that contains many existing algorithms.

if we ignore the redundant x at the inner loop levels.

In fact, every left-linear binary join plan $[R_1, R_2, R_3, \ldots]$ can be represented by a Free Join plan:

$$\begin{aligned}&[R_1(\mathbf{x_1}) \,|R_2(\mathbf{x_1} \cap \mathbf{x_2})]\\ \rightarrow\ &[R_2(\,\mathbf{x_2} \setminus \mathbf{x_1}) \mid R_3(\mathbf{x_2} \cap \mathbf{x_3})]\\ \rightarrow\ &[R_3(\mathbf{x_3} \setminus \mathbf{x_2}) \mid R_4(\mathbf{x_3} \cap \mathbf{x_4})]\\ &\vdots\end{aligned}$$

As we will see in Section 2.5, we can also represent any Generic Join plan with a Free Join plan. Free Join therefore generalizes and unifies both binary join and Generic Join. However, the true power of Free Join is its ability to represent algorithms like the one in Figure 4d that are neither binary join nor Generic Join. Intuitively, a (linear) binary join plan says which *relation* to process at each step, and always processes one additional relation at a time. A Generic Join plan says which *variable* to process at each step, and always processes one variable at a time. A Free Join plan can process any number of relations and variables at a time. As we show in Figure 5, this flexibilty allows Free Join to represent a much larger space of algorithms, leading to performance improvements beyond the existing algorithms.

2.5 Other Optimizations

In this section, we summarize a few additional optimizations introduced in our SIGMOD paper [15], as well as relate Free Join to the Generic Join algorithm. To motivate these optimizations, we will follow the new example query $Q_\triangle$ in Figure 7, and visualize an input instance in Figure 8. Note $Q_\triangle$ is now a *cyclic* query, because its hypergraph is a triangle with three vertices x, y, and z and three edges corresponding to $R(x, y)$, $S(y, z)$, and $T(z, x)$.

Let us first consider the algorithm in Figure 6a. We show the Free Join plan in the comments atop the figure, and note that it is equivalent to binary join. To reduce clutter we will stick with a row-wise notation while keeping in mind the underlying columnar storage. We also remove redudant values from the hash maps; for example, the hash map for S in Figure 6a now maps every y to a vector of z (instead of a vector of (y, z)). The binary join algorithm first iterates over (x, y)-tuples in R, using each y to probe into S to get a vector of z. For each z, it then probes into T to check if (z, x) is in T, and outputs the tuple (x, y, z) if so. This binary plan essentially computes the join $R \bowtie S$ first before discarding tuples that do not join with T. Because $R \bowtie T$ has size $\Omega(n^2)$, the algorithm runs in quadratic time. Since the relations are symmetric, any binary join plan will have the same asymptotic run time.

A different algorithm is shown in Figure 6b. Here, as we iterate over tuples in R, we look up into S and T at the same time. In other words we perform the late iteration optimization again, pulling up lookups to discard tuples early. However, this is not sufficient, because every tuple in R does in fact join with both S and T, so the first loop level discards no tuples. As the second loop level iterates over s, we are in effect still computing the join $R \bowtie S$ which takes $\Omega(n^2)$ time. To overcome this inefficiency, we now introduce a new operator called *intersection* ($\cap$).

2.5.1 Intersection

In contrast, the algorithm in Figure 6c runs in linear time. The small difference is that we have replaced the inner loop of Figure 6b, which iterates over s, with a loop iterating over the intersection of s and t. When we compute the intersection, we always iterate over the smaller set while probing into the larger set. To analyze the run time of this algorithm, we first assume we have built a hash map for S, mapping each y to a *hash set* of z, and similar for T. Building each hash map takes linear time. Then, as we iterate over R, we consider three cases:

1. For tuple (x_0, y_0), $s = S[y_0] = \{z_i \mid i \in [0, \ldots, n]\} = T[x_0] = t = s \cap t$, so we may iterate over either s or t to compute $s \cap t$ in linear time.

2. For each tuple $(x_0, y_i) \mid i > 0$, $t = \{z_i \mid i \in [0, \ldots, n]\}$ but $s = S[y_i] = \{z_0\}$, so we iterate over (the only element of) s and probe into t to compute $s \cap t$. This takes constant time for each y_i, so for all y_i the total time is linear.

3. The case for tuple (x_i, y_0) is symmetric to the previous case, so it also takes linear time.

Overall, the algorithm in Figure 6c runs in linear time. Another way to think about the intersection operation is to understand it as dynamically reording the iterations and lookups: to compute $s \cap t$, we switch between the plans $[S(z) \mid T(z)]$ and $[T(z) \mid S(z)]$ depending on which one is smaller.

2.5.2 Generic Join

We can now faithfully derive the Generic Join algorithm as a special case of Free Join, using the itersection operator. Suppose an instance of Generic Join follows the variable order $x_1, x_2, \ldots, x_n$. The corresponding Free Join plan is:

$$\begin{aligned}&[R_1^1(x_1) \cap R_1^2(x_1) \cap \cdots]\\ \rightarrow\ &[R_2^1(x_2) \cap R_2^2(x_2) \cap \cdots]\\ &\vdots\\ &\rightarrow [R_n^1(x_n) \cap R_n^2(x_n) \cap \cdots]\end{aligned}$$

where R_i^j is a relation that contains x_i in its schema. When computing a multiway intersection, we (dynamically) pick the smallest set to iterate over and probe into the rest. For any choice of the variable order, the Generic Join algorithm is guaranteed to run in worst-case optimal time [14, 9, 10].

```
# [ R(x, y) | S(y) ]
# -> [ S(z) | T(z, x) ]

for (x,y) in R:
  s = S[y]? # S:y->[z]
  for z in s:
    if (z,x) in T:
      output(x,y,z)
```

(a) Plan equivalent to binary join.

```
# [ R(x, y) | S(y), T(x) ]
# -> [ S(z) | T(z) ]

for (x,y) in R:
  s = S[y]?; t = T[x]?
  for z in s:
    if z in t:
      output(x,y,z)
```

(b) Another Free Join plan.

```
# [ R(x, y) | S(y), T(x) ]
# -> [ S(z) ∩ T(z) ]

for (x,y) in R:
  s = S[y]?; t = T[x]?
  for z in s ∩ t:
    output(x,y,z)
```

(c) Plan with intersect (∩).

```
# [ R(y) ∩ S(y) ]
# -> [ S(z) ∩ T(z) ]
#    -> [ T(x) ∩ R(x) ]
for y in R.y ∩ S.y:
  for z in S[y] ∩ T.z:
    for x in T[z] ∩ R[y]:
      output(x,y,z)
```

(d) Plan equivalent to Generic Join.

Figure 6: Four different Free Join plans for $Q_\triangle$ and their execution.

$$Q_\triangle \coloneq R(x,y), S(y,z), T(z,x).$$

```
SELECT * FROM R,S,T -- R(x,y), S(y,z), T(z,x)
 WHERE R.y = S.y AND S.z = T.z AND T.x = R.x
```

$$R = \{(x_0, y_i) \mid y \in [0 \dots n]\} \cup \{(x_i, y_0) \mid y \in [0 \dots n]\}$$
$$S = \{(y_0, z_i) \mid y \in [0 \dots n]\} \cup \{(y_i, z_0) \mid y \in [0 \dots n]\}$$
$$T = \{(z_0, x_i) \mid y \in [0 \dots n]\} \cup \{(z_i, x_0) \mid y \in [0 \dots n]\}$$

Figure 7: The triangle query $Q_\triangle$, and an input instance.

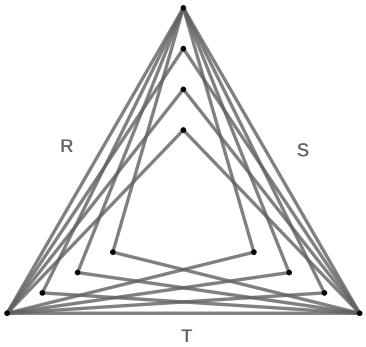


Figure 8: Visualization of input relations to $Q_\triangle$. Each relation is represented by a set of edges. The query $Q_\triangle$ looks for triangles formed by one edge from each relation (there are 10).

2.5.3 Lazy trie building

To explain the algorithm in Figure 6b we assumed to have pre-built hash maps for S and T. A more efficient strategy is to *lazily* construct parts of the data structures as we iterate over the relations. Specifically, we will only build the hash set for s (or t) right before we need to probe into it. This way, we can avoid building a linear number of (singleton) hash sets that we only need to iterate over. In the full paper [15] we describe a data structure, called Column-oriented Lazy Trie (COLT), that generalizes this idea.

2.5.4 Vectorized Execution

A simple way to implement Free Join is to use a recursive function, as shown in Figure 9. For every group in the Free Join plan, we iterate over the first subatom and probe into the remaining subatoms. If all probes are successful, we append new values to the partial tuple, and recursively call `join` on the remaining plan and sub-relations. This naïve implementation suffers from poor temporal locality: in the body of the loop, we probe into the same set of relations for each tuple. But these probes are interrupted by the recursive call at the end, which is itself a loop interrupted by further recursive calls.

```
def join(plan, tuple, R, S, T, ...):
  if plan is empty: output(tuple)
  else:
    let [ R(xs) | S(ys), T(zs), ... ] = plan[0]
    for xs in R:
      r = R[xs]?; s = S[ys]?; t = T[zs]?; ...
      join(plan[1:], tuple ++ xs, r, s, t, ...)
```

Figure 9: Recursive formulation of Free Join.

```
def join(plan, tuple, R, S, T, ...):
  if plan is empty: output(tuple)
  else:
    let [ R(xs) | S(ys), T(zs), ... ] = plan[0]
    tup_rels = {} # map a partial tuple to sub-relations
    for xs_batch in R.iter_batch():
      for xs in xs_batch:
        r = R[xs]?; s = S[ys]?; t = T[zs]?; ...
        tup = tuple ++ xs
        tup_rels[tup] = (r, s, t, ...)
    for (tup, rels) in tup_rels:
      join(plan[1:], tup, rels)
```

Figure 10: Vectorized execution for Free Join.

A simple way to improve locality is to perform a batch of probes before recursing, just like the classic vectorized execution for binary join. As shown in Figure 10, we call `iter_batch` to retrieve a batch of tuples from R. For each tuple in a batch, we probe into the relations to get the corresponding sub-relations. If all probes are successful, we append new values to the partial tuple, and pair the tuple with the respecitve sub-relations; otherwise we continue onto the next tuple in the batch. Finally, for each tuple that successfully probes into all relations, we call `join` recursively on the remaining plan.

3. EXPERIMENTS

We implemented Free Join as a standalone Rust library. The main entry point of the library is a function that takes a binary join plan (produced and optimized by DuckDB), and a set of input relations. The system converts the binary plan to a Free Join plan, optimizes it, then runs it using COLT and vectorized execution. We compare Free Join against two baselines: our own Generic Join implementation

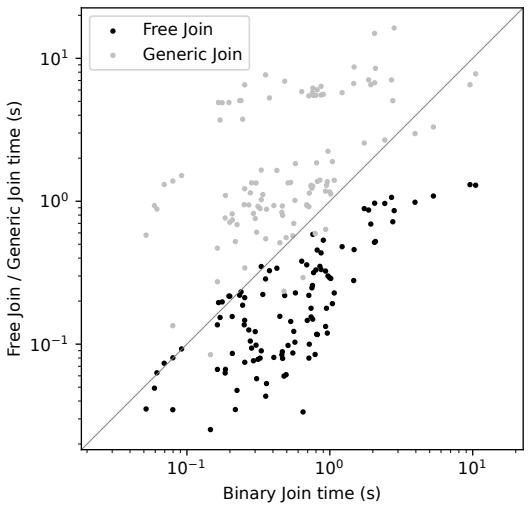


Figure 11: Run time comparison on JOB. Each black dot compares the run time of a query on Free Join and binary join, and a black dot below the diagonal means Free Join is faster. The gray dots compare Generic Join and binary join similarly.

in Rust, and the binary hash join implemented in the state-of-art in-memory database DuckDB [13, 12]. We evaluate their performance on the popular Join Order Benchmark (JOB) [7]. We refer the reader to our SIGMOD paper [15] for additional experiments that include other systems and benchmarks, as well as detailed ablation studies of the optimizations.

3.1 Setup

While we had easy access to optimized join plans produced by DuckDB, we did not find any system that produces optimized Generic Join plans, or can take an optimized plan as input. We therefore implement a Generic Join baseline ourselves, by modifying Free Join to fully construct all tries, and removing vectorization. We chose as variable order for Generic Join the same as for Free Join.[3]

The JOB benchmark contains 113 acyclic join queries with an average of 8 joins per query. Each query in the benchmarks only contains base-table filters, natural joins, and a simple group-by at the end, and no null values. The queries works over real-world data from the IMDB dataset. We exclude 5 queries that return empty results, since such empty queries are known to introduce reproducibility issues[4].

We ran all our experiments on a MacBook Air laptop with Apple M1 chip and 16GB memory. All systems are configured to run single-threaded in main memory, and we leave all of DuckDB's configurations to be the default. All systems are given the same binary plan optimzed by DuckDB. Since we are only interested in the performance of the join algorithm, we exclude the time spent in selection and aggregation when reporting performance. This excluded time takes up on average less than 1% of the total execution time.

[3]Free Join defines only a partial order; we extended it to a total order.

[4]See GitHub Issue #11: https://github.com/gregrahn/join-order-benchmark/issues/11

```
111101
JOIN
 |  \________9775
1919495      company_name(cid)
JOIN
 |  \________1
8123586      company_type(ctid)
JOIN
 |  \________1
24740873     info_type1(iid1)
JOIN
 |  \________1
148621556    info_type2(iid2)
JOIN
 |  \________1
177388547    kind_type(kid)
JOIN
 |  \________2609129
20885030     movie_companies(mid, cid, ctid)
JOIN
 |  \________1380035
 |            JOIN
 |             |  \_____1380035
 |            2528312   movie_info_idx(mid, iid1)
 |            title(mid, kid)
14835720
movie_info(mid, iid2)
```

Figure 12: Query plan produced by DuckDB for JOB Q13a. We show the join attributes next to each input relation, and label each relation with its size as well as each join with its (actual) cardinality.

3.2 Run time comparison

Figure 11 compares the run time of Free Join and Generic Join against binary join on JOB queries. We see that almost all data points for Free Join are below the diagonal, indicating that Free Join is faster than binary join. On the other hand, the data points for Generic Join are largely above the diagonal, indicating that Generic Join is slower than both binary join and Free Join. On average (geometric mean), Free Join is 2.94x faster than binary join and 9.61x faster than Generic Join. The maximum speedups of Free Join against binary join and Generic Join are 19.36x and 31.6x, respectively, while the minimum speedups are 0.85x (17% slowdown) and 2.63x.

We zoom in onto a few interesting queries for a deeper look. The slowest query under DuckDB is Q13a, taking over 10 seconds to finish. Generic Join runs slightly faster, taking 7 seconds, whereas Free Join takes just over 1 second. The query plan for this query, as shown in Figure 12, reveals the bottleneck for binary join: the first 3 binary joins are over 4 very large tables, and two of the joins are many-to-many joins, exploding the intermediate result to contain over 100 million tuples. However, all 3 joins are on the same attribute (`mid`); in other words they are quite similar to our clover query $Q_{\clubsuit}$. As a result, Generic Join and Free Join simply intersects the relations on that join attribute, expanding the remaining attributes only after other more selective joins.

On a few queries Free Join runs slightly slower than binary join, as shown by the data points over the diagonal. The binary plans for these queries are all bushy, and each query materializes a large intermediate relation. We have not spent much effort optimizing for materialization, and we implement a simple strategy: for each intermediate that we need to materialize, we store the tuples containing all

base-table attributes in a vector. Future work may explore more efficient materialization strategies, for example only materializing attributes that are needed by future joins.

4. CONCLUSION

In this paper we review the Free Join framework, which generalizes and unifies traditional join algorithms and WCOJ algorithms. We re-derive the more general Free Join from the well-understood binary join, hoping to make the framework more accessible to database practitioners. We hope to see the adoption of Free Join in mainstream databases, which shall inspire further research on the design of join algorithms. We conclude by pointing out some promising research directions. First, we have been focusing on single-threaded in-memory algorithms. How can we adapt Free Join to work on disk, on multi-core machines, and in distributed settings? In particular, the COLT data structure relies on laziness and appears inherently sequential. Do we need a new data structure to parallelize Free Join? Second, our optimizer starts from an already optimized binary plan, and conservatively improve it into a Free Join plan. How can we design and implement an optimizer to better exploit the flexibilty of Free Join? Finally, as the experiments show, our current implementation of Free Join is not yet competitive for certain bushy plans. How can we improve the performance of materializing intermediate results for Free Join?

5. REFERENCES

[1] D. J. Abadi, D. S. Myers, D. J. DeWitt, and S. Madden. Materialization strategies in a column-oriented DBMS. In R. Chirkova, A. Dogac, M. T. Özsu, and T. K. Sellis, editors, *Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, The Marmara Hotel, Istanbul, Turkey, April 15-20, 2007*, pages 466–475. IEEE Computer Society, 2007.

[2] C. R. Aberger, A. Lamb, S. Tu, A. Nötzli, K. Olukotun, and C. Ré. Emptyheaded: A relational engine for graph processing. *ACM Trans. Database Syst.*, 42(4):20:1–20:44, 2017.

[3] R. Fagin. Degrees of acyclicity for hypergraphs and relational database schemes. *J. ACM*, 30(3):514–550, 1983.

[4] M. J. Freitag, M. Bandle, T. Schmidt, A. Kemper, and T. Neumann. Adopting worst-case optimal joins in relational database systems. *Proc. VLDB Endow.*, 13(11):1891–1904, 2020.

[5] G. Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–170, 1993.

[6] 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 E. Sallinger, J. V. den Bussche, and F. Geerts, editors, *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2017, Chicago, IL, USA, May 14-19, 2017*, pages 429–444. ACM, 2017.

[7] V. Leis, A. Gubichev, A. Mirchev, P. A. Boncz, A. Kemper, and T. Neumann. How good are query optimizers, really? *Proc. VLDB Endow.*, 9(3):204–215, 2015.

[8] A. Mhedhbi and S. Salihoglu. Optimizing subgraph queries by combining binary and worst-case optimal joins. *Proc. VLDB Endow.*, 12(11):1692–1704, 2019.

[9] H. Q. Ngo. Worst-case optimal join algorithms: Techniques, results, and open problems. In J. V. den Bussche and M. Arenas, editors, *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, Houston, TX, USA, June 10-15, 2018*, pages 111–124, New York, NY, USA, 2018. ACM.

[10] 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, New York, NY, USA, 2012. ACM.

[11] H. Q. Ngo, C. Ré, and A. Rudra. Skew strikes back: new developments in the theory of join algorithms. *SIGMOD Rec.*, 42(4):5–16, 2013.

[12] M. Raasveldt. Duckdb - A modern modular and extensible database system. In S. R. Valluri and M. Zait, editors, *1st International Workshop on Composable Data Management Systems, CDMS@VLDB 2022, Sydney, Australia, September 9, 2022*, 2022.

[13] M. Raasveldt and H. Mühleisen. Data management for data science - towards embedded analytics. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org, 2020.

[14] T. L. Veldhuizen. Triejoin: A simple, worst-case optimal join algorithm. In N. Schweikardt, V. Christophides, and V. Leroy, editors, *Proc. 17th International Conference on Database Theory (ICDT), Athens, Greece, March 24-28, 2014*, pages 96–106, Athens, Greece, 2014. OpenProceedings.org.

[15] Y. R. Wang, M. Willsey, and D. Suciu. Free join: Unifying worst-case optimal and traditional joins. *Proc. ACM Manag. Data*, 1(2), jun 2023.

[16] M. Yannakakis. Algorithms for acyclic database schemes. In *Very Large Data Bases, 7th International Conference, September 9-11, 1981, Cannes, France, Proceedings*, pages 82–94. IEEE Computer Society, 1981.

Technical Perspective: Efficient and Reusable Lazy Sampling

Thomas Neumann
TUM
neumann@in.tum.de

When interactively working with data, query latency is very important. In particular when ad-hoc queries are written in an explorative manner, it is essential to quickly get feedback in order to refine and correct the query based upon result values. This interactive use case is difficult to support if the underlying data is large, as analyzing large volumes of data is inherently expensive.

An attractive way to tackle this problem is to use approximate query processing (AQP). Instead of computing the exact query result, the system produces an approximate answer to the query, which is often good enough when still interactively exploring the data, and sometimes even good enough as the final answer [1]. The advantage of using approximate answers is that these can be computing much more efficiently, sometimes orders of magnitude faster than the exact result. And if the user is only interested in a rough overview over the data the full precision of, e.g., aggregate values is not required anyway.

Approximate query processing is usually based upon sampling techniques, that is the query is evaluated not on the full data set but on a random sample of data [2], which is much smaller but which exhibits the same data distribution as the original data. For simple queries like

```
select avg(x) from R
```

that is straight forward, the query will produce roughly the same result when executing on a random sample of `R` instead of the full table. But when the query contains filter predicates like

```
select avg(x) from R where y<4
```

the situation becomes more difficult, as a random sample might contain no or only a few tuples that satisfy the filter condition.

To alleviate that systems have mainly two options: Either they use larger samples, which makes it less likely that they are unable to answer the query, but which increases the AQP evaluation time and the storage costs. Or they use stratified sampling, which means that they maintain samples for a given predicate (or a given set of values).

Which allows for answering a query if a suitable sample is present, even for selective predicates, but which makes samples less versatile. For example a sample for the condition `y<4` can also be used to answer queries with a predicate `y<3`, but not to answer queries with `y<6`. For very predictable and repetitive queries that is less of an issue, but for the interactive use cases that is a severe problem, as queries can vary greatly.

Having one large sample over everything does not work well for selective predicates, but computing a sample for every predicate that we see in a query is not very practical, as the number of combinations is very large and eagerly computing samples is expensive.

The next paper tackles this problem by an interesting observation: We can construct a larger uniform random sample from two smaller uniform random samples over the same domain if 1) both samples come from disjoint parts of the original relation, and 2) we know how large the original data partitions were. Basically we can union two existing samples into a larger one, similar to a reservoir sampling strategy, but we have to take into account the selection probabilities for the elements were different. This allows for flexible stitching together available samples such that the query can be answered with the available data, even if the samples do not perfectly match the query.

There are more technical hurdles that have to be overcome, and the system must make a decision which sampling should be maintained given both the query and the already existing sample, but for that read the paper.

1. REFERENCES

[1] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. Blinkdb: queries with bounded errors and bounded response times on very large data. In Z. Hanzálek, H. Härtig, M. Castro, and M. F. Kaashoek, editors, *Eighth Eurosys Conference 2013, EuroSys '13, Prague, Czech Republic, April 14-17, 2013*, pages 29–42. ACM, 2013.

[2] A. Birler, B. Radke, and T. Neumann. Concurrent online sampling for all, for free. In D. Porobic and T. Neumann, editors, *16th International Workshop on Data Management on New Hardware, DaMoN 2020, Portland, Oregon, USA, June 15, 2020*, pages 5:1–5:8. ACM, 2020.

Efficient and Reusable Lazy Sampling

Viktor Sanca
EPFL
viktor.sanca@epfl.ch

Periklis Chrysogelos*
Oracle
periklis.chrysogelos@oracle.com

Anastasia Ailamaki
EPFL
anastasia.ailamaki@epfl.ch

ABSTRACT

Modern analytical engines rely on Approximate Query Processing (AQP) to provide faster response times than the hardware allows for exact query answering. However, existing AQP methods impose steep performance penalties as workload unpredictability increases. While offline AQP relies on predictable workloads to a priori create samples that match the queries, as soon as workload predictability diminishes, returning to existing online AQP methods that create query-specific samples with little reuse across queries results in significantly smaller gains in response times. As a result, existing approaches cannot fully exploit the benefits of sampling under increased unpredictability.

We propose LAQy, a framework for building, expanding, and merging samples to adapt to the changes in workload predicates. We propose lazy sampling to overcome the unpredictability issues that cause fast-but-specialized samples to be query-specific and design it for a scale-up analytical engine to show the adaptivity and practicality of our framework in a modern system. LAQy speeds up online sampling processing as a function of data access and computation reuse, making sampler placement after expensive operators more practical.

1 Reducing the Data Volume with Sampling

Data exploration is crucial to deriving insights and informed decisions in today's data-driven world. Visualization tools and interactive dashboards provide a convenient and rich exploration interface. Nevertheless, humans require fast responses to maintain their focus during mental tasks: Miller [22] reports a 0.1 seconds response window for users to feel the UI follows their actions and 15 seconds as a hard limit to avoid demoralization and breaking the line of thought. However, with current memory technologies providing a few hundred GBps of memory bandwidth, analytical engines cannot even process simple queries that touch more than a few GBs of data in the 0.1 seconds window, despite running on TB-sized memories.

Analytical engines have long relied on approximate query processing to reduce the data processing time [2, 19, 6, 3, 27, 24, 8, 5, 29, 15]. Offline approximate query processing methods prebuild samples to reduce the data access and processing time during query execution [2]. However, the significant savings of such approaches come at the expense of requiring predictable workloads. Such predictable workloads appear in data warehousing operations, but they mismatch interactive workloads like data exploration, where queries constantly change [18]. Online query processing generalizes offline sampling into interactive use cases. Instead of relying on query templates, it does the sampling during query execution before expensive operations. Such approaches reduce the processing time; however, sampling during query execution is a heavy operation [29, 3]. To minimize this cost, existing approaches rely on pushing lightweight, selective operations below sampling [15] and sample caching [24]. However, specializing the online sample to the current query reduces its potential to be suitable for another query. As a result, existing approaches introduce a steep trade-off between sample reuse and sampling overhead for interactive exploration.

This work introduces LAQy, a sampling-based AQP framework that increases the sample reuse opportunities while maintaining the sample creation to the same or lower cost than online sample creation. LAQy relaxes the sample matching requirements by allowing samples to match a query partially. This relaxation allows using samples that would otherwise be considered inappropriate for an incoming query – resulting in significant execution time savings. LAQy corrects the query-sample mismatch using *delta* samples: samples that LAQy uses to augment the partially matching sample with the missing pieces needed to satisfy the query approximation requirements. As a result, LAQy extends i) the effectiveness of online sampling techniques to a greater range of query workloads, ii) provides a system-level acceleration technique that maintains the theoretical sample properties, and iii) reduces the query predictability requirements that existing systems need to overcome the hardware limitations through query approximation techniques.

Overall, LAQy improves the efficiency of online approximate query processing systems by increasing sample reuse and bridging the gap between predictable and unpredictable predicates in approximate query processing. As a result,

*Work done while the author was at EPFL

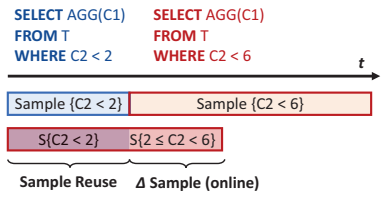


Figure 1: Relaxing the predicate predictability allows reuse.

LAQy enables fast responses despite the query unpredictability that characterizes data exploration workloads, which previously hindered the effectiveness of query approximation methods. While most prior related systems use sampling in disk-based, single-threaded, and distributed setups, we investigate and address the bottleneck shift in scale-up single-machine setups with minimal engine modifications. LAQy proposes a lazy sampling algorithm to avoid costly sample creation through an in-memory-friendly architecture and judicious workload-driven sample construction and merging. This allows for speeding up the sampling operation for base relations and also enables reuse opportunities when samplers must be placed after joins, for example, when meaningful filtering and sampling dimensions are only available after joining the fact tables with dimension tables.

2 Online Flexibility and Offline Predictability

Exploratory data analysis creates hard-to-predict query patterns, yet its interactive nature requires fast response times to keep the data analyst focused and productive. Further, keeping the user engaged requires response times exceeding the underlying hardware's capabilities. As a result, analytical engines have relied on approximate query processing to reduce the data processing cost. A core requirement of AQP methods is that when the query implies some grouping, all groups are represented in the output. To achieve that, variants of approximation methods often rely on stratified sampling: the systems analyze the query clauses and extract the columns that, if not included in the stratification key, the query output could sample out tuples. These columns are called the *Query Column Set* (QCS), and when aligned with the query requirements, they allow for strict bounds on the error of the computed aggregations [2]. Furthermore, Quickr [15] provides optimization rules for injecting the samplers in the query plan and transforming samples and their QCS requirements while pushing them across various relational operators. Conversely, the remaining non-QCS columns are called *Query Value Set* (QVS).

Workload predictability. The applicability of the different approximate query processing methods depends on workload predictability. Agarwal et al. [2] classify workloads into four categories based on the predictability of the queries: i) *predictable queries* where the upcoming queries are known, e.g., monthly or weekly warehousing tasks that repeat the same query in a fixed interval, ii) *predictable query predicates* where the *filter conditions* of upcoming queries are fixed and known, iii) *predictable QCSs* where the *grouping* or *filtering* columns are known, but the actual filtering values are revealed only during the query invocation, and iv) *unpredictable queries* where there is no information about the upcoming queries.

While the query patterns during data exploration are hard to predict, they are also not entirely unpredictable. Users often add, remove, expand, or shrink filters, grouping columns and joined tables to focus their exploration on specific subsets of the input or expand to increase their exploration scope and hypothesis testing [30]. As a result, the analytical engine often receives incremental changes to the query shape, placing such workloads between predictable, or slowly changing, QCSs and predictable query predicates. Yet, approximation techniques for predictable query predicates are much more efficient as samples are handled similarly to offline views and thus incur small overheads. In contrast, (mostly) predictable QCSs rely on online sampling.

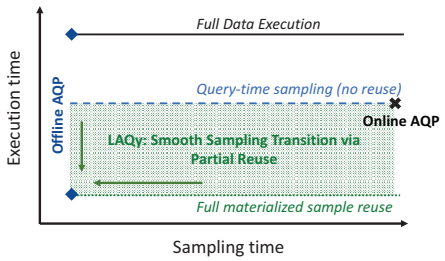


Figure 2: The design space of sampling-based AQP methods.

Issue #1: strict sample matching rules. Whether a sample subsumes a query or not is, currently, a binary classification: either a query has an appropriate QCS and QVS, or it can not provide the necessary guarantee, even if just one column is not contained in one of those sets. As a result, small mismatches in column set requirements create a new sample – with the corresponding overhead and missed opportunities.

Example. Figure 1 shows two queries arriving in a sequence, with the second having an expanded filter, similar to the queries submitted by a data scientist who zooms out to cover a greater input range. Suppose that, during query execution, a sample on T is created after the filtering condition, e.g., to minimize the sampling cost. That sample will not have the information necessary to answer the expanded (red) query. As a result, while the two queries are very similar, the sample would be rejected, and a new one would be created, reducing the first sample's effectiveness.

Challenge #1: increasing the sample usefulness. Efficient query approximation techniques are needed to increase the usefulness of each sample by relaxing the sample matching requirements despite the theoretical requirements. To avoid compromising the theoretical sample properties, LAQy increases the sample usefulness by allowing samples to partially satisfy a query and creating delta queries that require smaller samples for their approximation.

Issue #2: creating a new sample is costly. When existing samples do not match the current query, a new sample is created, or approximate processing is abandoned altogether. To mitigate such overheads, existing methods [24] aggressively cache samples and synopsis. Specifically, Taster [24] continuously inspects the workload to materialize samples as a side-effect of execution for future (re)use based on the recent query history. However, the materialized sample is only helpful if it entirely subsumes the predicates and QCS, otherwise falling back to the online AQP processing. The new sample is built from scratch despite potentially having similar samples. As a result, despite an existing sample sat-

isfying part of the query, the new sample will be built on the entire input and have the full QCS size, which prior work has shown can have a prohibitive cost, especially as the QCS and QVS sizes increase [29].

Example. If the queries of Figure 1 were also grouped on C2, then the corresponding samples would have C2 in the QCS. Although the blue sample is a subset of the sample required for the red query, existing approaches would discard the blue sample when evaluating the second (red) query. Further, with the valid range of C2 increasing, building the second (red) sample with stratification on C2 would be significantly slower than making the blue sample, as the number of strata is the dominating factor during sample creation [29].

Challenge #2: efficiently reacting to missing samples. Efficient query approximation techniques are needed to build the required samples quickly and efficiently. To reduce the sample creation time, LAQy judiciously builds only the missing parts of a required sample and retrieves the rest from the already available samples.

Issue #3: unpredictability increases the risk of useless pre-sampling. Offline sampling [2] mitigates the sampling overhead by sampling before the query time. During query processing, the sample is already available, effectively eradicating the cost of sampling from the response time. Further, to avoid the overhead of periodically recreating the sample as the source data are updated, Birler et al. [3] propose incrementally maintaining the sample during updates. Yet, offline methods require that the useful samples are known before query time. To reduce the probability of wasted samples, offline approaches tend to build more inclusive and, thus, bigger samples. However, this is still a double-edged sword. While it may support more queries, 1) using a bigger sample incurs a higher cost for each query using it, 2) a higher build cost, and yet the potential of mispredicting the query workload and not using the sample.

Example. Consider this time that an aggressive sampling approach sees the condition on C2, and instead of applying the filter before the sampling, it decides to stratify on C2 as well (include C2 in the QCS). While this would make the sample reusable in the second (red) query, it may not pay off if, for example, multiplying the number of strata by the cardinality of C2 makes sample creation much slower than creating two different samples – a common occurrence for high cardinality columns [29].

Challenge #3: sampling without regret. As a result, the over-generalization of samples may result in significant costs. Efficient query approximation systems need to minimize the risk of wasteful sample creation by reducing the probability of creating a minimally useful sample. To overcome this issue, LAQy builds only samples that will be immediately used to accelerate a query.

Summary. The current rigidness of whether a sample satisfies a query, combined with the high cost of sample creation and the uncertainty about the long-term gains of creating bigger samples, highly affect the effectiveness of online approximation techniques. Further, data exploration often results in partially overlapping queries and small transitions across predicates, resulting in the aforementioned issues for such workloads. We describe how LAQy's design allows tackling all three issues through a design that focuses on maximizing the (even partial) sample reuse to minimize the sample creation (through lazy delta samples) and a sample-as-you-query model that reduces the regret for building a sample by creating only samples that are immediately useful. As a result, LAQy bridges the gap between online and offline approximation methods by partial sample reuse (Figure 2).

3 The Design Principles of Lazy Sampling

LAQy is an approximate query processing engine designed to bridge the gap between offline and online AQP methods in scale-up systems. Existing approaches have to select between building low-overhead samples specialized to the query predicates or paying a higher sample construction overhead to create reusable samples. Instead, LAQy achieves a sweet spot between reusability and sampling time by taking advantage of the mergeable nature of samples. Furthermore, LAQy is compatible with adaptive storage and sample budgeting solutions, like Taster [24], to allow adaptive management of the allocated sample space based on workload patterns.

Instead of imposing a binary can-or-cannot-be reused per-sample decision, the samples generated by LAQy create a continuous spectrum between online and offline sampling methods (Figure 2). As a result, samples do not have to be entirely ignored; instead, they provide a partial input to the query. Furthermore, for the query input that is not covered by the existing sample, LAQy avoids building a full sample. Instead, it constructs only the part of the sample necessary for the current query – minimizing the overhead imposed by sample construction, resulting in better applicability of our method in the era of in-memory engines that saturate the available memory bandwidth.

The three core principles of LAQy enable it to directly address the challenges outlined in Section 2:

Principle #1: Partial reuse. Predicates are traditionally considered either known or too volatile to specialize the sample to the predicate value – creating a steep penalty, even if the predicates are not entirely random. For example, typical data exploration patterns [30] imply that the focus of interest may change but correlate to the initial scope of analysis, albeit in unpredictable ways. LAQy exploits the overlap across the predicates to reuse the existing materialized samples and compute only the delta samples to satisfy uncovered, non-overlapping sample ranges, extending the strategy of offline sampling systems.

Principle #2: Judicious sampling. Stratified sampling is an expensive operation. Stratification imposes random accesses making sampling potentially as expensive as joins or aggregations. Nonetheless, it reduces the input to upcoming operations and the execution time of future queries if the sample is materialized and reused. To avoid excessive overheads without a corresponding long-term speedup, LAQy minimizes the input of stratification operations to the minimum required for the current query, along the direction of the online sampling systems.

Principle #3: Laziness and minimal waste. Every bit counts, but some bits count more. The long-term gain of a sample depends on upcoming queries. While the prediction accuracy of future queries can vary, samples created for the current query have an immediate turnaround. Also, we can give a higher turnaround to already created samples by increasing their utility through partial reuse. LAQy reduces the

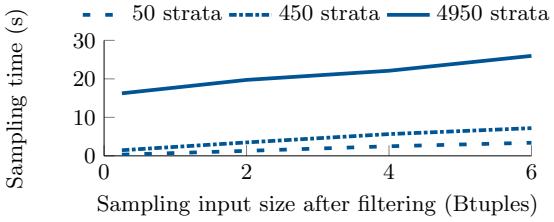


Figure 3: Impact of #tuples of the dataset and #strata defined by the QCS on the stratified sample creation time.

decision-making and pushes it as late as possible by building only the necessary delta samples and merging them only when needed. This makes our approach complementary to both the online and offline state-of-the-art sampling-based AQP systems.

4 Pruning the Sample Construction Space

LAQy bridges the gap between offline and online sampling by exploiting the overlap of query predicates to reuse and merge samples. Section 4.1 starts by evaluating the different parameters that affect the build time for a stratified sample. Section 4.2 links the various parameters with the query predictability. Lastly, in Section 4.3, it motivates our main observation: relaxing the predictability requirements for predicates allows lazily building samples and amortizes the (stratified) sample build time across queries with overlapping predicates.

4.1 Building Stratified Samples

To create a stratified sample, each input element is inserted into a reservoir based on the element values on the QCS columns. Each stratum thus keeps track of the reservoir and the number of considered elements (*weight*) as the admission probability depends on the number of previously considered items.

Considering an input item for inclusion into a reservoir generally requires random access to find and update the *admission-control* state (*random number generator*). If the item is admitted, then finding and replacing an existing item from the reservoir requires one more random access. The latter may or may not be on the same cache line as the admission state, depending on the reservoir capacity (k) and whether the reservoir's storage is inlined with the admission state.

Admission control occurs for every tuple, so the corresponding time is reduced as the input tuples decrease. Admission, however, happens stochastically with a probability that converges to the sampling rate as more items are considered per reservoir. As a result, admissions are responsible for a small portion of the build time. Furthermore, following the observation that an item will be infrequently replaced, LAQy does not pack the reservoir storage with the admission state. Instead, it stores a pointer to the actual reservoir together with the admission state to reduce the footprint of the hash table data structure used to store the strata [29].

Figure 3 shows how the build time increases with an increase in the input size. Independently of the number of strata, the sampling time follows a similar trend. However, as the number of strata grows, we observe a higher sampling time, even for small input sizes. Each stratum introduces a

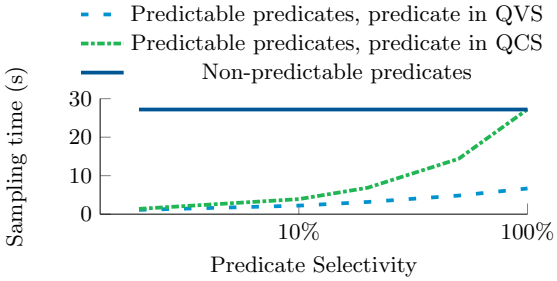


Figure 4: Stratified sampling time for various selectivities.

constant allocation and initialization even if it does not reach capacity. As a result, while the input size affects the sample building time, strata initialization time, count, and random accesses to the stratum state significantly impact the sample build time, exacerbated by the total cardinality of QCS.

Key observations & predicate predictability. LAQy builds on top of two observations. First, the number of strata and tuples has the highest impact on sampling time, while per-stratum capacity k has a lower impact on the build time. Second, increases in the reservoir capacity have a marginal impact on the overall sampling time while controlling the desired error bounds by obtaining sufficient sample support. LAQy takes advantage of these two observations to accelerate sampling for unpredictable predicates by exploiting patterns and overlaps present in the query predications that reduce the penalty of mispredictions in the case of partial predicate matches.

4.2 The Cost of Predicate Unpredictability

Suppose we have the following query that we would like to approximate by creating a sample, where the input relation T can be a base table or a subquery result.

```
SELECT C1, SUM(C2) FROM T
WHERE C3 > 5 [AND C1 IN ('G1', 'G2')]
GROUP BY C1
```

If the predicate values are predictable on `C3` and `C1`, the future queries are assumed to satisfy those conditions. This means that building a stratified sample on QCS=`{C1}` with filters pushed down before the sampler can answer the future queries with equal or *stricter*, subsuming predicates[1].

If the predicate values are not predictable, but predicate columns are, those columns are added to the QCS without pushing down the filters before sampling. This makes the resulting sample applicable to any predicate on those columns at the cost of a larger input and more complex stratified sampling with multiple columns in QCS. As the cost of sampling is not negligible at query execution time, the cost of more complex sampling schemes adds a prohibitive penalty. The user may be interested only in limited, yet unpredictable, dataset regions, incurring wasteful precomputation at a critical execution path. Suppose this happens if the user is interested only in specific groups in the future queries with a filter on `C1` in brackets.

To demonstrate the impact of selecting one of the above options, Figure 4 shows the sample build time required for

[1]If a sufficient sample size support exists for requested error guarantees after applying the stricter predicate [10, 2, 15].

creating a stratified sample for different selectivities. Section 6 provides the detailed input and hardware details. We start with a stratified sample on QCS={C1} and filter pushdown on C3 (*predictable predicates on QVS*), which results in 450 strata. To improve the reusability of the materialized sample without any predictions about the runtime predicate values, the column C3 is added to QCS without any filter pushdown, resulting in 4950 strata (*non-predictable predicates*).

However, when the predicate is on a column C1 that would participate in the QCS due to being part of the GROUP BY clause, the filter can be pushed down along with adding the column to the QCS. In the best case, on the highly selective end for the newly added QCS, this approach prunes the sample input instead of making a strict decision to add the entire column to the QCS (*predictable predicate on QCS*). There is up to 19-24x slowdown to create a non-predicate-specific sample, representing an average slowdown of 6.7-11x across the selectivities to resolve the predicate value unpredictability with the **existing all-or-none sample matching** systems and approaches. The goal is to construct a reusable sample as the highest speedup is achieved using a previously materialized sample (offline AQP) instead of online sampling. The leading cause of the predicate predictability rigidity is the strict requirement of predicate subsumption for reuse. **The sample reuse dichotomy** creates a clear-cut tradeoff between reusability and performance, driven by predictions instead of the actual workload with *overlapping* predicates.

4.3 Relaxing Sample Predicate Predictability

There is, however, a middle ground between predictable and unpredictable predicates. Consider the two queries in Figure 1 to demonstrate this. If the value for predicate on C2 is known ahead of time, the query belongs in the predictable predicate category. If that value varies, then the predicate would be considered unpredictable. Suppose a sample is materialized as a side-effect of workload. In that case, the sample can be reused if the predicates are subsumed [24], assuming enough tuples in the sample satisfy the predicate to achieve the desired error guarantees.

To our best knowledge, existing systems and algorithms opt to rebuild samples if the predicate is not fully subsumed, even if generating an online sample for the missing range $[2, 6)$ would suffice to answer the query using the previously materialized, *effectively offline* sample, as in Figure 1.

We call a sample generated for the uncovered range $(x, y]$ a Δ *(delta)* sample, the process of generating such sample *expansion* as it expands the samples' predicate coverage, and the process of combining the reusable sample with Δ sample *merging*. We call the overall sampling *lazy*, as it defers and relaxes the strict predictability rules as late as possible while reducing the work that samplers need to do.

Summary. Our approach relaxes the predictability requirements to improve the reuse of previously materialized samples through workload-aware methods described by prior work [2, 24]. We modify the physical sampling algorithm and obtain an equivalent *logical* sample with the requested characteristics. We explore the reservoir-based sampling algorithms (simple reservoir sample, stratified reservoir sampling) commonly used in systems, but mainly since *merging* two or multiple reservoirs preserve the characteristics of the final (reservoir) sample [33]. We propose a method that bridges *offline* and *online* sampling, **complementary** to the prior art.

5 Lazy Sampling with LAQy

Traditionally [2, 15], the columns in a stratified sample are split into two sets: the QCS (*Query Column Set*) and the QVS (*Query Value Set*). The QCS contains the columns that affect the participation of the rows in the output, such as columns participating in filters, join conditions, and grouping columns. The QVS has the remaining columns, such as the aggregation columns.

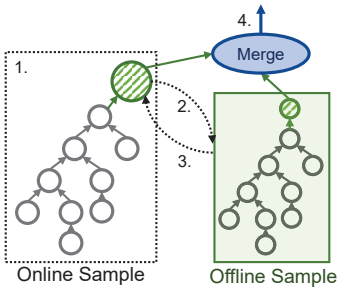


Figure 5: LAQy: the relaxed sample reuse framework and steps.

We describe the main steps of how LAQy relaxes the predicate requirement for sample reuse, depicted in Figure 5:

1. Starting from an approximable query, an optimizer (e.g. [15, 24]) decides on the *logical* sampler placement (striped green circle). The sampler can have as input base relations or subqueries in a more general case.
2. The sample store attempts to find a materialized sample with the requested characteristics of the *logical* sampler.
   (a) If there is a sample that already covers the predicate space with a subsuming predicate, we can use the existing sample if sufficient sample support exists after filtering, fully reusing the *offline sample*.
   (b) If no sample exists overlapping in predicates and requirements, *online sampling* is performed only.
   (c) If a partially overlapping sample exists, calculate the non-overlapping predicate for the Δ query and prepare the sample for merging in the query execution pipeline. Proceed to the next step (3).
3. The predicates are pushed down the original query plan to initiate online Δ sampling.
4. Online and offline (stratified) reservoirs are merged, producing the equivalent logical sample.

LAQy combines samples and triggers the generation of a new partial (delta) sample only for the relevant input data not covered by the existing samples. Thus, for each sample, LAQy maintains the *Query Predicate*, *Query Input*, *QCS* and *QVS* that represent the (*logical*) input of each sampler (the striped circle, Offline Sample in Figure 5). Making this information part of the sample description makes explicit that the sample is malleable and reusable based on specific conditions outlined in the rest of this section.

To combine two samples, LAQy relies on selecting samples that cover the area of interest but do not overlap. Consider a query predicated on $C3 > 2$. It can be served by combining a sample on $C3 > 5$ with a sample on $C3 \in (2, 5]$. However, combining a sample on $C3 > 5$ with a sample on $C3 \in (2, 7]$ would sample tuples in the $C3 \in (5, 7]$ range twice, introducing a bias towards those intervals – violating the per-reservoir uniformity assumption. To resolve this, LAQy *extends* samples by pushing down the predicates to merge reservoirs on non-overlapping predicates in QVS, reducing the sampler input as a desirable property of Δ samples.

LAQy relies on prior art that analyzes and proposes solutions for optimization rules for sampling for online AQP [15], as well as workload-based predictions and sample materializations [2, 24]. We aim to bridge and relax the dichotomy between the two approaches via flexible sample reuse. Therefore, we focus on a solution that synergetically works with the prior art and present and evaluate our contributions within reasonable starting assumptions. At either extreme, our system would run as purely *online* or *offline* sampling-based AQP system. For example, we will assume that an offline sample exists to focus on the behavior and performance of the proposed relaxed reuse via lazy sampling.

5.1 Sample Merging

Reservoir sampling is amenable to updates and maintenance while preserving the requested sample characteristics. We use the property of well-defined *merge* behavior to enable both the independent, data-parallel scale-up execution as well as the key contribution that relies on sample merging. Different data processing operations, such as filtering or stratification, impact the data distribution. Further, partitioning and/or splitting the data for parallelization potentially introduces additional data skew. As a result, using samples created after such operations requires careful consideration to avoid creating biased samples.

LAQy uses the weighted reservoir sampling algorithm [4] to recover a sample equivalent to a full resample of the input. Specifically, during merging, LAQy weighs the elements of the to-be-merged samples based on what proportion of the input they represent. To do so, each created sample keeps the reservoir R and tracks the running sum of (importance) weights w of previously qualifying elements. This allows LAQy to use the exact reservoir weights as it creates and stores samples just in time, effectively maintaining the count of the elements as the weight of each qualifying input element is one. Using these (importance) weights, LAQy calculates the new weights associated with every sample to use them further with the weighted reservoir sampling algorithm so that the elements of the merged sample have the same weight as after a full resample.

The key observation is that two independent reservoirs with their associated weights $\{R_1, w_1\}$ and $\{R_2, w_2\}$ can be *merged* to obtain $\{R_m, w_1 + w_2\}$. Intuitively, reservoir R_1 represents w_1 tuples, R_2 represents w_2 tuples and a reservoir R_m equivalent to sampling the union of original input data of R_1 and R_2 would have combined weights $w_1 + w_2$. To avoid resampling the original input data, we use the existing samples where we adjust the sampling weight from uniform to biased, as in general case weights are different, where elements of R_1 are selected with probability $\frac{w_1}{w_1+w_2}$; where converse holds for elements of R_2 [33]. More formally, to combine reservoirs R_1 and R_2 into R_m, we perform weighted reservoir sampling [4] with R_1 and R_2 and their weights as input to the algorithm. The result is equivalent to performing reservoir sampling over the combined input while avoiding repeating processing and accessing the original data represented by R_1 and R_2. Therefore, merging itself does not change the properties of the obtained reservoir as it is a reformulation of the inputs to the algorithm that is amenable to changing the input weights. Query-driven sampling and operator placement, even past expensive operations, results in samples where the error estimation framework is compatible with principles established by prior art [8, 15, 10].

Since, in general, even the reservoir sizes k may differ, we introduce *Scaled Proportional Sampling* to further bias the weight by the ratio of $k1$ and $k2$, obtaining the weight factor of $\frac{k_{scaled}}{w}$ to achieve proportional reservoir merge via weighted reservoir sampling of the inputs. LAQy maintains reservoirs with their weights, and the reservoir sizes are parameters known at runtime.

5.2 Issuing Δ Samples with Relaxed Predicates

LAQy differentiates between two general types of reuse across the samples: tightening and relaxing the predicates depending on the available materialized samples that may require issuing a Δ sampling query.

Conditional transition to stricter predicates happens when a sample already covers the wanted space, but the predicate of interest is stricter. We can use the existing sample under the condition that samples have enough support after the predicate pushdown to satisfy the accuracy requirements [24]. No Δ sample needs to be issued if the sample meets the condition; otherwise, a complete online sampling must be performed to satisfy the guarantees.

Relaxing predicates requires adding input tuples for predicate values not covered by an existing sample. We issue a Δ query based on the predicates to find the *inverted*, non-overlapping interval.

Combined tightening and relaxing is the case when predicates require tightening of the predicates on an existing *offline* sample and relaxing via executing the Δ sample for the missing interval. However, the sample support (of each stratum) after applying the predicate $\sigma_{predicate}(S)$ is unknown ahead of time, which may impact the (specified) expected error bounds. If strict qualifying sample support is needed, we can follow the conservative approach: an online query is subsequently executed for all the reservoirs/strata that do not have sufficient support, which performs sampling after the filter pushdown. This validates if the *exact* low (or lack of) support comes from the original data distribution or as the artifact of sampling. We can continue the query in a less strict setting and report the obtained error bound with the available data. One approach to reducing the probability of insufficient sample support is introducing an oversampling factor $\alpha \geq 1$ to create reservoirs sized $\alpha \cdot k$. This trades-off space for tentatively higher sample reusability in case of strict predicates, similar to how past approaches [2] create samples with different parameters k.

Predicate relaxing allows for building only a Δ sample on the missing value range. LAQy reduces the input to the expensive sampling operations by minimizing the necessary work at query time. We reduce the wasteful processing while taking a sampling decision as late as possible to process only the data of interest to the user, maximizing the reuse of previously materialized *online* or *offline* sampling effort.

6 Evaluating LAQy in a Scale-Up Engine

We implement LAQy in Proteus [28, 16, 7], a parallel in-memory DBMS engine. Proteus uses LLVM to generate customized code for each query, as in the prior work on JIT engines [23, 7, 29, 21]. As Proteus did not have AQP support, we extended it by 1. introducing sampler operators with the corresponding code generation routines, 2. adding a sample lifetime management module that captures the generated samples to allow reuse on subsequent queries, and 3. implementing sample merging aggregation functions and operators designed for scale-up execution.

Hardware setup. We run our experiments on a server with dual-socket Intel(R) Xeon(R) Gold 5118 CPU (2x12 cores, 2x24 threads, HyperThreading enabled) with 384GB DDR4 RAM. All the experiments run on 48 threads.

Dataset. We use the Star Schema Benchmark [25] data for our experiments, using the scale factor (SF) 1000 in a binary columnar layout. We add a unique identifier column (`lo_intkey`) to the `lineorder` table as an 8-byte integer value ranging from 0 to the number of elements in the table, randomly shuffled to enable fine-grained selectivity control without implying a specific data ordering. The necessary columns are preloaded in memory for the experiments.

In line with findings from Microsoft's production big-data cluster [14], where authors report that 90% of the input column sets have between 1 and 6 columns, for our evaluation, we use 1 to 3 columns to produce up to 4950 strata.

Workload. We use two representative query templates to evaluate the holistic performance of online sampling in LAQy. Equation (Q1) describes a scan-heavy query where the sampler is pushed down to the scan operator. Equation (Q2) describes a query with a random-access pattern due to joins with dimension tables that enable sampling and stratifying on other meaningful dimensions, where the sampler is placed higher in the query plan. For both queries, we control and report the selectivity over the fact table (`lo_orderdate`) via the `lo_intkey` attribute.

```
SELECT AGG() FROM lineorder
WHERE lo_intkey BETWEEN(lower AND upper)   (Q1)
GROUP BY lo_orderdate
```

We generate two query sequences to simulate an exploratory workload with changing predicates. The first sequence represents a `long-running` analysis: the user is running the specified query template over 50 iterations such that they progressively extend the value range and narrow it down or use the same interval at a specified rate r. The second sequence represents when the user changes the focus of interest during their analysis, where 60 queries are split into 3x20 batches that we name `short-running` analyses, which are similar in setup to the `long-running` sequence.

```
SELECT AGG() FROM lineorder, date, supplier, part
WHERE lo_intkey BETWEEN(lower AND upper)
AND s_region="AMERICA" AND p_category="MFGR#12"
AND ...  (JOIN) GROUP BY (d_year,p_brand1)
                                           (Q2)
```

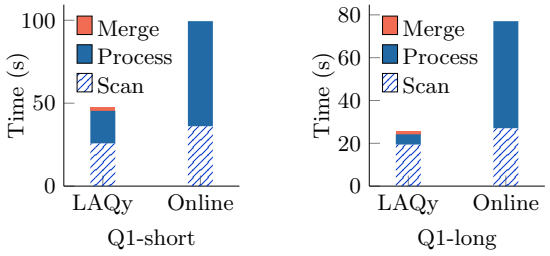
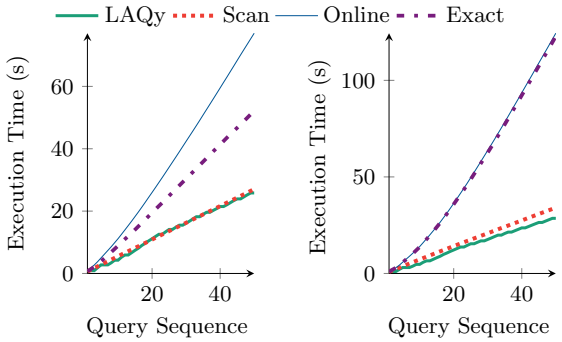


Figure 6: Cumulative processing time breakdown.



(a) Q1 pattern (b) Q2 pattern

Figure 7: Long query sequence, cumulative execution time.

6.1 Reducing the Cost of Online Sampling

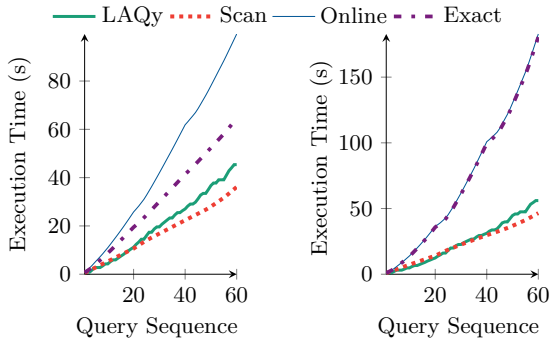
LAQy benefits from reuse opportunities due to overlapping predicates, which enable lazy sampling speedup. In concrete terms, we first present the breakdown of the cumulative processing time of Q1 in Figure 6. First, scan time is lower in the case of LAQy due to detecting full sample reuse opportunities. Second, the processing time (excluding scan) is lower due to lazy sampling and issuing only required Δ samples. Finally, there is a negligible merge overhead to produce an equivalent sample since merging the reservoirs operates over the data samples.

6.2 Efficient & Reusable Sampling with LAQy

In case there is a good workload prediction, offline sampling methods have the potential to create a reusable set of samples. In case the workload is unpredictable or evolving, our approach reduces the misprediction penalty by creating additional samples only over the queries and data relevant to the user - which is definitely known at runtime. We conclude the experimental analysis of LAQy by observing the big picture of cumulative execution time.

6.2.1 *Long-running sequence: high reuse*

Lazy sampling can achieve the cost effectively below the scan time for samplers pushed down to the base relations (Figure 7a). This is due to combining offline sample use that does not require a scan and progressively creating missing sample components. Equally, this could be the case of a good workload prediction where slight predicate deviations occur that our approach can gracefully fix. In a scan-heavy query, regular online sampling is slower than the exact counterpart, as it introduces sample processing overhead on top of the input coming in at a memory-bandwidth rate (scan).



(a) Q1 pattern (b) Q2 pattern

Figure 8: Short query sequence, cumulative execution time.

Similarly, when the sampler is placed further up in the query plan as in Figure 7b, the combined savings from full offline reuse and selectivity-driven partial reuse enable expensive online sampling costs to be reduced to a scan. Online sampling cost is the same as the exact execution since the input to both operators has a lower throughput due to preceding joins, where additional online sampling processing is masked with the input data rate.

6.2.2 *Short-running sequence: adapting to change*

With moderate reuse opportunity, LAQy enables lazy sampling between the scan and the input rate defined by the preceding operator(s). In particular, when the sampler is fully pushed down, the lazy sampling enables cumulative execution time comparable to sequential scan (Figure 8a). Otherwise, the cumulative cost of lazy sampling is between the scan and the input rate of input operators, reducing both the impact of random data access patterns of preceding operations and the sampling cost through delta sampling and predicate pushdown synergetically, as we show in Figure 8b.
Takeaway. LAQy enables sampling methods to convert the dichotomy of reuse based on full match into a flexible spectrum based on partial matching. As a result, LAQy expands the applicability of existing sampling schemes to a greater query set with a minimal processing overhead.

7 Related Work

Approximate query processing has been of interest over the years as a method to trade-off accuracy for reduced execution times and has found applications in interactive data exploration [18], data visualization [17], cardinality estimation for query optimization [11], and in novel execution schemes such as speculative execution [31, 13].

The prior work and research in the field of AQP is vast and covers the theory, systems, operators, and approximation schemes [8], well summarized in a survey [19]. We limit the scope of our paper to sampling-based approximations in the context of modern scale-up analytical systems, and we briefly present the related work.
Offline AQP systems [26, 2, 12, 20] build samples and data summaries ahead of time, based on assumptions of knowing the future workload and static or slowly changing data. However, when an adequate sample is unavailable, fallback to regular execution or online sampling is often necessary, reducing the expected speedups. LAQy uses the previously created samples to avoid extensive sample creation and to reduce the size of new samples. As a result, LAQy's continuous sample creation increases the potential of having the required sample ready and thus brings online approximation methods closer to the benefits of offline ones.
Online AQP systems [15, 14] perform sampling at runtime and offer speedup through data reduction by injecting samplers at data-intensive parts of the query plan. As the original data needs to be processed, such systems provide lower speedups than their offline counterparts. Furthermore, they do not exploit workload patterns that may speed up future queries through sample reuse. In contrast, LAQy's aggressive sample caching methodology reduces the overhead of online sampling to only online sampling for the newly created delta sample.
Hybrid AQP systems combine online and offline AQP techniques. Taster [24] proposes adapting to the workload by materializing offline samples relevant to the recent window of queries and otherwise performing online approximations. Taster makes a coarse-grained decision about full sample matches and does not explore the opportunity of partial sample reuse. LAQy exploits the overlap between required (unpredictable) and materialized (predictable) sample sets in exploratory workloads to reduce the sample creation time.
Model updates and concept drifts. The ubiquitous nature of volatile data has motivated research in machine learning and theoretical approaches to detect and update out-of-date models [9, 32, 1, 33, 4] where new data are monitored to detect when the input data distribution has significantly shifted from the maintained model. When this happens, the online model is retrained or incrementally updated.

8 Conclusion

The reuse dichotomy between the predictable predicates used by offline sampling to speculate on the useful samples and the online sampling, which pays the performance cost of unpredictability in ad-hoc queries, imposes a steep performance penalty, even if an overlapping sample exists. We propose bridging this gap by relaxing the sample reuse requirements and allowing partial sample reuse. We increase the utility of the samples created using the assumption of predictability and pay only the necessary cost to perform online sampling due to the predicate unpredictability. Our lazy sampling approach adapts to the changes in query predicates and improves performance proportionally to the reuse savings. In effect, lazy sampling allows flexible data access and computation reuse without reprocessing the full, original data input. As the data volume continues to grow, and in conjunction with modern, efficient, and hardware-conscious data management systems, to enable faster and more interactive analytical queries or speculative execution opportunities, judicious and reuse-aware sampling becomes an ever-important part of efficient approximations in modern analytical engines.

9 Acknowledgments

We thank the reviewers and the shepherd of the original SIGMOD'23 paper, and our colleagues and friends at the DIAS Lab at EPFL for their insightful comments, suggestions, and valuable feedback. This work was partially funded by the Swiss National Science Foundation (SNSF) project "Efficient Real-time Analytics on General-Purpose GPUs" (178894).

10 References

[1] P. K. Agarwal, G. Cormode, Z. Huang, J. M. Phillips, Z. Wei, and K. Yi. Mergeable summaries. 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 23–34. ACM, 2012.

[2] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. Blinkdb: Queries with bounded errors and bounded response times on very large data. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, page 29–42, New York, NY, USA, 2013. Association for Computing Machinery.

[3] A. Birler, B. Radke, and T. Neumann. Concurrent online sampling for all, for free. In *Proceedings of the 16th International Workshop on Data Management on New Hardware*, DaMoN '20, New York, NY, USA, 2020. Association for Computing Machinery.

[4] M. T. Chao. A general purpose unequal probability sampling plan. *Biometrika*, 69(3):653–656, 12 1982.

[5] S. Chaudhuri, G. Das, and U. Srivastava. Effective use of block-level sampling in statistics estimation. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, page 287–298, New York, NY, USA, 2004. Association for Computing Machinery.

[6] S. Chaudhuri, B. Ding, and S. Kandula. Approximate query processing: No silver bullet. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, page 511–519, New York, NY, USA, 2017. Association for Computing Machinery.

[7] P. Chrysogelos, M. Karpathiotakis, R. Appuswamy, and A. Ailamaki. Hetexchange: Encapsulating heterogeneous CPU-GPU parallelism in JIT compiled engines. *Proc. VLDB Endow.*, 12(5):544–556, 2019.

[8] G. Cormode, M. N. Garofalakis, P. J. Haas, and C. Jermaine. Synopses for massive data: Samples, histograms, wavelets, sketches. *Found. Trends Databases*, 4(1-3):1–294, 2012.

[9] J. a. Gama, I. Žliobaitundefined, A. Bifet, M. Pechenizkiy, and A. Bouchachia. A survey on concept drift adaptation. *ACM Comput. Surv.*, 46(4), mar 2014.

[10] P. B. Gibbons, V. Poosala, S. Acharya, Y. Bartal, Y. Matias, S. Muthukrishnan, S. Ramaswamy, and T. Suel. Aqua: System and techniques for approximate query answering. Technical report, Technical report, Bell Labs, 1998.

[11] H. Harmouch and F. Naumann. Cardinality estimation: An experimental survey. *Proc. VLDB Endow.*, 11(4):499–512, Dec. 2017.

[12] B. Hilprecht, A. Schmidt, M. Kulessa, A. Molina, K. Kersting, and C. Binnig. Deepdb: Learn from data, not from queries! *Proc. VLDB Endow.*, 13(7):992–1005, Mar. 2020.

[13] S. Igescu, V. Sanca, E. Zapridou, and A. Ailamaki. Improving k-means clustering using speculation. 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. Zaït, 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.

[14] S. Kandula, K. Lee, S. Chaudhuri, and M. Friedman. Experiences with approximating queries in microsoft's production big-data clusters. *Proc. VLDB Endow.*, 12(12):2131–2142, Aug. 2019.

[15] S. Kandula, A. Shanbhag, A. Vitorovic, M. Olma, R. Grandl, S. Chaudhuri, and B. Ding. Quickr: Lazily approximating complex adhoc queries in bigdata clusters. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 631–646, New York, NY, USA, 2016. Association for Computing Machinery.

[16] M. Karpathiotakis, I. Alagiannis, and A. Ailamaki. Fast queries over heterogeneous data through engine customization. *Proc. VLDB Endow.*, 9(12):972–983, 2016.

[17] A. Kim, E. Blais, A. Parameswaran, P. Indyk, S. Madden, and R. Rubinfeld. Rapid sampling for visualizations with ordering guarantees. *Proc. VLDB Endow.*, 8(5):521–532, Jan. 2015.

[18] T. Kraska. Approximate query processing for interactive data science. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, page 525, New York, NY, USA, 2017. Association for Computing Machinery.

[19] K. Li and G. Li. Approximate query processing: What is new and where to go? - A survey on approximate query processing. *Data Sci. Eng.*, 3(4):379–397, 2018.

[20] Q. Ma and P. Triantafillou. Dbest: Revisiting approximate query processing engines with machine learning models. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 1553–1570, New York, NY, USA, 2019. Association for Computing Machinery.

[21] P. Menon, A. Ngom, T. C. Mowry, A. Pavlo, and L. Ma. Permutable compiled queries: Dynamically adapting compiled queries without recompiling. *Proc. VLDB Endow.*, 14(2):101–113, 2020.

[22] R. B. Miller. Response time in man-computer conversational transactions. In *American Federation of Information Processing Societies: Proceedings of the AFIPS '68 Fall Joint Computer Conference, December 9-11, 1968, San Francisco, California, USA - Part I*, volume 33 of *AFIPS Conference Proceedings*, pages 267–277. AFIPS / ACM / Thomson Book Company, Washington D.C., 1968.

[23] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *Proc. VLDB Endow.*, 4(9):539–550, June 2011.

[24] M. Olma, O. Papapetrou, R. Appuswamy, and A. Ailamaki. Taster: Self-tuning, elastic and online approximate query processing. In *35th IEEE*

International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019, pages 482–493. IEEE, 2019.

[25] P. O'Neil, E. O'Neil, X. Chen, and S. Revilak. *The Star Schema Benchmark and Augmented Fact Table Indexing*, page 237–252. Springer-Verlag, Berlin, Heidelberg, 2009.

[26] Y. Park, B. Mozafari, J. Sorenson, and J. Wang. Verdictdb: Universalizing approximate query processing. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 1461–1476, New York, NY, USA, 2018. Association for Computing Machinery.

[27] J. Peng, D. Zhang, J. Wang, and J. Pei. Aqp++: Connecting approximate query processing with aggregate precomputation for interactive analytics. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 1477–1492, New York, NY, USA, 2018. Association for Computing Machinery.

[28] A. Raza, P. Chrysogelos, A. G. Anadiotis, and A. Ailamaki. Adaptive HTAP through elastic resource scheduling. In D. Maier, R. Pottinger, A. Doan, W. Tan, A. Alawini, and H. Q. Ngo, editors, *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, pages 2043–2054. ACM, 2020.

[29] V. Sanca and A. Ailamaki. Sampling-based AQP in modern analytical engines. In *DaMoN*, pages 4:1–4:8. ACM, 2022.

[30] B. Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *Proceedings of the 1996 IEEE Symposium on Visual Languages*, VL '96, page 336, USA, 1996. IEEE Computer Society.

[31] P. Sioulas, V. Sanca, I. Mytilinis, and A. Ailamaki. Accelerating complex analytics using speculation. In *CIDR*, 2021.

[32] A. Tahmasbi, E. Jothimurugesan, S. Tirthapura, and P. B. Gibbons. Driftsurf: Stable-state / reactive-state learning under concept drift. In M. Meila and T. Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, volume 139 of *Proceedings of Machine Learning Research*, pages 10054–10064. PMLR, 2021.

[33] C. Wyman. *Ray Tracing Gems II: Next Generation Real-Time Rendering with DXR, Vulkan, and OptiX*, chapter 22, Weighted Reservoir Sampling: Randomly Sampling Streams, pages 345–349. Apress, Berkeley, CA, 2021.

Technical Perspective: Unicorn: A Unified Multi-Tasking Matching Model

AnHai Doan
Department of Computer Science
University of Wisconsin-Madison, USA
anhai@cs.wisc.edu

Data integration has been a long-standing challenge for data management. It has recently received significant attention due to at least three main reasons. First, many data science projects require integrating data from disparate sources before analysis can be carried out to extract insights. Second, many organizations want to build knowledge graphs, such as Customer 360s, Product 360s, and Supplier 360s, which capture all available information about the customers, products, and suppliers of an organization. Building such knowledge graphs often requires integrating data from multiple sources. Finally, there is also an increasing need to integrate a massive amount of data to create training data for AI models, such as large language models.

Integrating data from different sources often requires matching: deciding whether two data elements are "semantically the same", where "data elements" can be strings, tuples, columns, etc. Numerous matching tasks have been studied. Well-known examples include string matching, entity matching (which matches tuples), schema matching (which matches columns), ontology matching, entity linking, entity alignment, and column type annotation.

Over the past 40 years, these matching tasks have been intensively studied in many research communities (including databases, AI, Web, Semantic Web, and KDD). Significant progress has been made, and today there is a general consensus on the overall architecture to solve these tasks.

Consider the problem of matching two sets A and B, which contain data elements of the same type (such as strings, tuples, or columns). State-of-the-art solutions typically solve this problem in two steps: *blocking* and *matching*. Considering all pairs in the Cartesian product of A and B is often impractical, so the blocking step uses heuristics to quickly discard pairs ($a \in A, b \in B$) that are unlikely to match (e.g., people living in different cities). The matching step focuses on the remaining pairs, and applies a rule- or learning-based matcher to each pair to predict match/no-match.

The highlighted Unicorn paper focuses on the matching step. It starts with the observation that, despite decades of work, today within an organization people are still likely to solve the matching tasks *separately*. For example, they may use three separate labeled data sets to train three matchers to match strings, tuples, and columns, respectively.

Inspired by recent progress in deep learning, the paper asks: can we *unify* these tasks, i.e., train a single matcher to match all kinds of data elements, such as strings, tuples, and columns, as well as new unseen types of data elements?

The paper shows that this is indeed possible, and describes an elegant solution architecture based on recent progress in deep learning. To match a pair ($a \in A, b \in B$), the solution first serializes the pair into a textual sequence, then uses a pre-trained language model to map the sequence into an embedding vector. Next, the solution transforms the vector using a neural-network mixture-of-expert model, then applies a binary classifier to output a probability that the input pair is a match. The parameters of the neural-network based architecture are trained using the labeled data sets of "all" available matching tasks. Extensive experiments with 20 data sets over seven matching tasks show the promise of the proposed solution.

Thus, the Unicorn paper introduces the first solution that can solve the matching step of a wide variety of matching tasks. The paper correctly observes that this can enable learning across the matching tasks (because the solution is trained on the union of the labeled data sets of all tasks), and that the solution can be used to solve new matching tasks where no labeled data is yet available (a.k.a., zero-shot matching).

In addition, the paper raises a number of interesting questions. First, does this mean we now just need a single matching solution per organization? Probably not, because organizations often have use cases that require the data to be labeled differently. For example, two products may be declared a match in the context of customer service, yet not a match in the context of marketing. Yet, the possibility of being able to unify most matching solutions is intriguing and worth exploring further.

Second, even if we assume that an organization still needs multiple matching solutions, there still seems to be significant value in developing a single solution that can match a wide variety of tasks and can perform zero-shot matching. Perhaps this solution can be quickly customized (e.g., fine tuned) to help match a particular use case. How to do this is a topic of potentially great interest to both academia and industry.

Finally, while the paper has demonstrated the promise of a single unifying solution, how to make such a solution scalable and accurate for a very large amount of data – a scenario that is very common in practice – is still unclear, and is likely to motivate a lot of follow-up work.

Unicorn: A Unified Multi-Tasking Matching Model

Ju Fan[1], Jianhong Tu[1], Guoliang Li[2], Peng Wang[1], Xiaoyong Du[1]
Xiaofeng Jia[3], Song Gao[3], Nan Tang[4]*
[1]Renmin University of China, Beijing, China [2]Tsinghua University, Beijing, China
[3]Beijing Big Data Centre, Beijing, China [4]HKUST (GZ) / HKUST, Guangzhou, China
fanj@ruc.edu.cn

ABSTRACT

Data matching, which decides whether two data elements (*e.g.,* string, tuple, column, or knowledge graph entity) are the "same" (*a.k.a.* a match), is a key concept in data integration. The widely used practice is to build task-specific or even dataset-specific solutions, which are hard to generalize and disable the opportunities of knowledge sharing that can be learned from different datasets and multiple tasks. In this paper, we propose `Unicorn`, a unified model for generally supporting common data matching tasks. Building such a unified model is challenging due to heterogeneous formats of input data elements and various matching semantics of multiple tasks. To address the challenges, `Unicorn` employs one generic `Encoder` that converts any pair of data elements (a, b) into a learned representation, and uses a `Matcher`, which is a binary classifier, to decide whether a matches b. To align matching semantics of multiple tasks, `Unicorn` adopts a mixture-of-experts model that enhances the learned representation into a better representation. We conduct extensive experiments using 20 datasets on 7 well-studied data matching tasks, and find that our unified model can achieve better performance on most tasks and on average, compared with the state-of-the-art specific models trained for ad-hoc tasks and datasets separately. Moreover, `Unicorn` can also well serve new matching tasks with zero-shot learning.

1. INTRODUCTION

Data matching generally refers to the process of deciding whether two data elements are the "same" (*a.k.a.* a match) or not, where each data element could be of different classes such as string, tuple, column, etc. Data matching, as a core concept in data integration [10] and data preparation [7], includes a wide spectrum of tasks. In this paper, we consider common data matching tasks, including entity matching, entity linking, entity alignment, string matching, column type annotation, schema matching, and ontology matching.

Due to their importance, almost all the aforementioned matching tasks have been extensively studied in both database and machine learning communities, and still remain to be important research topics. Traditional wisdom to solve the matching tasks is to build task-specific or even dataset-specific (machine learning) models [27, 24, 8, 42], which are referred to as *specific models*. However, these specific models do not facilitate knowledge sharing across different tasks or datasets. For example, the knowledge learned by a column type annotation model (*e.g.,* TURL [8]) cannot be shared with an entity matching model (*e.g.,* Ditto [24]) due to different neural network designs, although very likely the two models can help each other. Moreover, training (or fine-tuning) each model for each task or dataset usually needs a huge amount of labeled data, which is time and effort consuming to obtain. For example, DeepMatcher [27] and Ditto [24] have to be fine-tuned on a new training dataset with hundreds of labeled matching/non-matching pairs.

In this paper, we propose `Unicorn`, a unified model to support multiple data matching tasks, as shown in Figure 1. Compared with the previous specific models, `Unicorn` has the following key innovations.

1. ***Task unification*** that generalizes task-specific solutions into one unified model, thus achieving lower maintenance complexity and smaller model sizes, compared with specific models.

2. ***Multi-task learning*** [44, 6] that enables the unified model to learn from multiple tasks and multiple datasets to make full use of *knowledge sharing*, which even outperforms specific models trained only on their own datasets separately.

3. ***Zero-shot prediction*** [31, 40] that allows the model to make predictions for a new task or a new dataset with *zero* labeled matching/non-matching pairs.

Although several unified models have been recently studied in the machine learning community [1, 33, 34], building such a unified model for supporting multiple data matching tasks is challenging. Firstly, data elements in the matching tasks can take ***heterogeneous formats***, such as tuples and columns in tabular data, entities in knowledge graphs, and plain text, which will increase the difficulty of task unification. Secondly, each data matching task may have its ***unique matching semantics***, making it non-trivial to enable knowledge sharing among multiple tasks.

*Corresponding author.

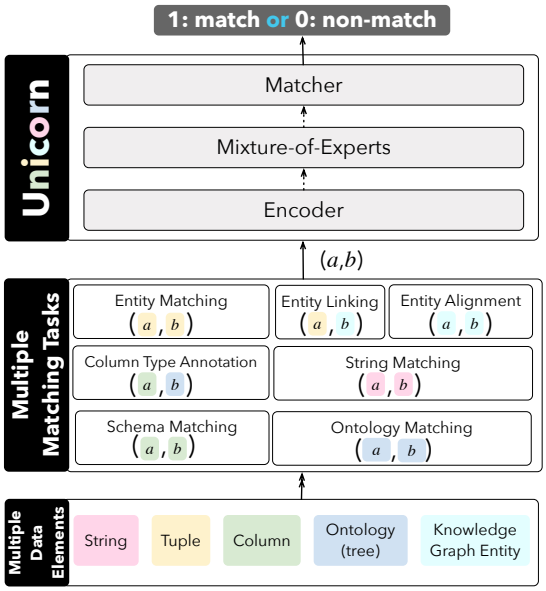


Figure 1: `Unicorn` is a unified model for "data matching" task in data integration. So far, it supports seven matching tasks in data integration for a pair (a, b) where a or b may be of the same or different types of data elements.

To address the challenges, we develop a general framework consisting of three key modules: an `Encoder`, a `Mixture-of-Experts` layer, and a `Matcher` (see Figure 1). The `Encoder` converts any pair of elements (a, b) with heterogeneous formats into a learned representation. Specifically, it serializes any pair of elements into text while still preserving their inherent structure, and employs a unified pre-trained language model for effective encoding. The `Mixture-of-Experts` layer enhances the learned representation into a better representation, in order to align matching semantics of various tasks or datasets. The `Matcher` predicts either 1 (for match) or 0 (for non-match) by taking the above representation as input.

Contributions. We make the following contributions.

(1) As far as we know, `Unicorn` is the first unified multi-tasking matching model for data integration. We formally define the problem of data matching tasks and introduce an overview of the `Unicorn` framework.

(2) We develop effective techniques for two key modules in the `Unicorn` framework, *i.e.,* a unified representation learning method for the `Encoder` module, and effective methods for the `Mixture-of-Experts` layer.

(3) Extensive experiments using 20 datasets for the 7 common data matching tasks show that `Unicorn`, as a unified model, outperforms the state-of-the-art specific models on most tasks and on average.

(4) We make a unified benchmark for multiple data matching tasks available at Github. We publish the source code of `Unicorn`, and provide the trained `Unicorn` model with multi-task learning in Hugging Face, so that researchers and practitioners can either use it out-of-the-box, or fine-tune it for specific tasks.

`https://github.com/ruc-datalab/Unicorn`
`https://huggingface.co/RUC-DataLab/unicorn-plus-v1`

2. PROBLEM: DATA MATCHING TASKS

2.1 Data Elements

In this paper, we consider a *data element* as a basic unit of information in data integration, which has a unique meaning. We consider five types of data elements.

String. A *string* is a sequence of words, which could be a noun or a natural language sentence, denoted as $\langle \texttt{word}_i \rangle_{1 \le i \le k}$.

Tuple. A *tuple* is a row contained in a table, consisting of a set of attribute-value pairs $\{(\texttt{attr}_i, \texttt{val}_i)\}_{1 \le i \le k}$, where $\texttt{attr}_i$ and $\texttt{val}_i$ are respectively the i-th attribute name and value of the tuple.

Column. A *column* is a set of values $\{\texttt{val}_i\}_{1 \le i \le k}$ of a particular attribute `attr` within a table, one value for each row of the table.

Ontology. An *ontology* identifies and distinguishes hierarchical concepts and the relationships among the concepts. An ontology is formalized as a *tree* structure $\texttt{ont} = \{\texttt{node}_i, \texttt{pnode}_i\}_{1 \le i \le k}$, where $\texttt{pnode}_i$ is the parent of $\texttt{node}_i$. The unique node, which does not have a parent node, is called the *root* node.

Knowledge Graph Entity. A *knowledge graph entity* (or *KG-entity* for short) describes a real-world entity, such as people, places, and things, in a knowledge graph. Formally, a knowledge graph (KG) is defined as a *graph* structure $\texttt{KG} = (E, R, A, V, T_r, P_a)$, where $\texttt{ent} \in E$, $\texttt{rel} \in R$, $\texttt{attr} \in A$, and $\texttt{val} \in V$ represent an entity, a relation, an attribute, and an attribute value respectively. Moreover, $(\texttt{ent}_i, \texttt{rel}_k, \texttt{ent}_j) \in T_r(\texttt{ent}_i)$ denotes a relational triple, and $(\texttt{attr}_k, \texttt{val}_j) \in P_a(\texttt{ent}_i)$ denotes an attribute-value pair.

2.2 Data Matching

Let $A = \{a_1, \ldots, a_n\}$ and $B = \{b_1, \ldots, b_m\}$ be two sets of data elements. The problem of *data matching* is to find all the pairs $(a_i, b_j) \in A \times B$ that are *matched*, where the semantic of whether (a_i, b_j) is matched depends on the specific data matching tasks. We consider the following seven common types of data matching tasks. For each task, we describe the existing solutions and our formal definitions.

(1) **Entity matching** refers to the task of determining whether two different tuples from two tables refer to the same real-world object [13, 5], which is formalized as determining whether a (`Tuple`, `Tuple`) pair matches or not.

(2) **Entity linking** refers to the task of determining whether a mention in a table refers to the same object with a `KG-Entity` in a knowledge base. A mention is represented by a tuple that contains multiple attributes. Formally, given a pair of (`Tuple`, `KG-Entity`), one needs to deduce whether they are matched or not.

(3) **Entity alignment** refers to a task of determining whether a pair of two KG-entities (`KG-Entity`, `KG-Entity`), typically from different knowledge graphs (*e.g.,* DBPedia and YAGO), are the same real-world object.

(4) **String matching** refers to the task of determining whether two strings (`String`, `String`) from two data sources are semantically the same or not.

(5) **Column type annotation** determines the semantic types of a column in a table, which is formalized as determining whether a (`Column`, `Ontology`) pair matches or not.

(6) **Schema matching** determines the correspondences between columns of two schemata from different tables. A typical schema matching solution requires to determining whether a (`Column`, `Column`) pair matches or not.

(7) **Ontology matching** finds correspondences between semantically related entities from different ontologies, formalized as determining if an (`Ontology`, `Ontology`) pair matches.

3. A UNIFIED MATCHING FRAMEWORK

3.1 An Overview of Unicorn

`Unicorn` offers a unified approach to multi-tasking data matching, characterized by two main advantages:

(1) **Task unification** standardizes task-specific solutions into a unified framework, achieving lower development complexity, smaller model sizes and easier adaption of new tasks.

(2) **Multi-task learning** [44, 6] enables the possible opportunities of *knowledge sharing* among different data matching tasks compared with training each task separately.

As depicted in Figure 2, `Unicorn` unifies multiple data matching tasks into a *text-to-prob* format, offering flexibility and extensibility in handling diverse tasks. It achieves this through input serialization of any data element pair (a, b) into a text sequence and outputs a probability $\hat{y}$ indicating the likelihood of a match. `Unicorn` comprises three core modules: `Encoder`, `Mixture-of-Experts`, and `Matcher`, each playing a crucial role in the unified data matching process.

Input: Multiple Data Matching Tasks. Instead of considering an individual data matching task, `Unicorn` takes as input a collection of data matching tasks, denoted as $\mathcal{T} = \{T_i\}$. In particular, each task $T_i = (A_i, B_i, \tau_i, \mathcal{D}_i)$ is composed of two sets of data elements to be matched, *i.e.,* A_i and B_i, and τ_i is the type of task T_i (see Section 2 for the supported task types). $\mathcal{D}_i \subset A_i \times B_i \times \{0, 1\}$ is a set of labeled examples, each of which denotes whether a pair of elements $a \in A_i$ and $b \in B_i$ is a match (*i.e.,* label 1) or a non-match (*i.e.,* label 0). Figure 2 (a) shows three example data matching tasks, *i.e., entity matching* over A_1 and B_1, *entity linking* over A_2 and B_2 and *schema matching* over A_3 and B_3. Note that `Unicorn` is extensible that new task types can be easily supported.

`Encoder`: The Input Layer. Given an element pair (a, b) from any data matching task (*e.g.,* Figure 2 (a)), the aim of `Encoder` is to first serialize the pair into a *text sequence* x, and then map x into a high-dimensional vector-based representation $\mathbf{x}$, *i.e.,*

$$\mathbf{x} = \mathsf{F}(x) = \mathsf{F}(\mathsf{S}(a, b)), \tag{1}$$

where $\mathsf{S}(\cdot)$ is a generic function for serializing any data element pair (a, b) from the matching tasks in $\mathcal{T}$ into a text sequence, and $\mathsf{F}(\cdot)$ is a pre-trained language model (PLM) for deriving high-dimensional feature vectors from the serialized sequences. See Section 3.2 for details.

`Mixture-of-Experts`: The Intermediate Layer. Although all the pairs from different tasks are mapped into one feature space, the distributions of their representations may not be aligned, as shown in Figure 2 (b). Consequently, it is hard to train a good `Matcher`.

To address the problem, we introduce an intermediate layer `Mixture-of-Experts` (MoE) [26, 35, 32, 22] to align the representations of different tasks, such that a good `Matcher` is easier to learn, as shown in Figure 2 (c). The basic idea is to transform original features of the pairs into an aligned feature space, *i.e.,* $\mathbf{x}' = \mathsf{MoE}(\mathbf{x})$. See Section 3.3 for details.

Matcher: The Output Layer. Given the representation $\mathbf{x}'$, the `Matcher` is a binary classifier, which takes a vector $\mathbf{x}'$ as input and outputs its probabilities $\hat{y}$ of matching. For `Matcher`, MLP is the most common choice used in existing deep learning-based classifier (*e.g.,* DeepER [14] and Ditto [24]), which is denoted as $\hat{y} = \mathsf{M}(\mathbf{x}')$.

Multi-task Training. We adopt multi-task supervised learning [44, 6] to train `Unicorn`, using labeled match/non-match examples coming from all tasks in $\mathcal{T}$. Specifically, we union all the labeled element pairs in $\mathcal{T}$ to generate a training set $\mathcal{D} = \bigcup_i \mathcal{D}_i$, and train the above three modules of `Unicorn` in an end-to-end manner.

Data Matching Prediction. After multi-task training over all the tasks in $\mathcal{T}$, `Unicorn` can support the following prediction scenarios.

(1) Unified Prediction on Existing Tasks. In this scenario, we use `Unicorn` to predict the test set $\mathcal{D}_i^{\mathsf{tst}}$ of **any** task $T_i \in \mathcal{T}$. Note that $\mathcal{D}_i^{\mathsf{tst}}$ is *unlabeled* and disjoint with the training set $\mathcal{D}_i$ of T_i.

(2) Zero-Shot Prediction on New Tasks. We can also use `Unicorn` to predict a **new** task T, which is not included in $\mathcal{T}$, with a *zero-shot* setting [31, 40] such that the pairs in the new task have **zero** labels.

3.2 The Encoder Module

Our approach is inspired by recent developments in *unified* NLP frameworks. We propose serializing all data matching tasks into *text* format and then using a unified PLM for encoding. This section aims to determine the optimal format for unifying different matching tasks and identify the most effective *unified* PLM, overcoming the limitations of using different PLMs for various tasks.

3.2.1 Pair-to-Text Serialization

PLMs are natural choices for encoders, which typically take a sequence of tokens as input. Hence, we propose to serialize a pair of data elements (a, b) into a sequence of tokens (like Ditto [24]) as:

$$x = \mathsf{S}(a, b) = \texttt{[CLS]}\ \mathsf{S}(a)\texttt{[SEP]}\ \mathsf{S}(b)\ \texttt{[SEP]} \tag{2}$$

where `[CLS]` is a special token to indicate the start of the sequence, the first `[SEP]` is a special token to separate the sequence of element a (*i.e.,* $\mathsf{S}(a)$) and the sequence of element b (*i.e.,* $\mathsf{S}(b)$), and the last `[SEP]` token is used to indicate the end of the sequence.

Next, we will describe how to serialize each type of the data elements.

String serialization. Given a string `str` with words $\langle \texttt{word}_i \rangle_{1 \le i \le k}$, we use the WordPiece tokenization algorithm, like BERT [9], to serialize `str` into a sequence of sub-words (tokens) as $\mathsf{S}(\texttt{str}) = \texttt{token}_1\ \texttt{token}_2\ \ldots\ \texttt{token}_{k'}$.

Tuple serialization. Given a tuple `tup` with attribute-value pairs $\{(\texttt{attr}_i, \texttt{val}_i)\}_{1 \le i \le k}$, we serialize it into a sequence as

$$\mathsf{S}(\texttt{tup}) = \texttt{[ATT]}\ \texttt{attr}_1\ \texttt{[VAL]}\ \texttt{val}_1 \ldots \texttt{[ATT]}\ \texttt{attr}_k\ \texttt{[VAL]}\ \texttt{val}_k,$$

where `[ATT]` and `[VAL]` are two special tokens for specifying attributes and values respectively.

Column serialization. Given a column `col` with an attribute name and values $(\texttt{attr}, \{\texttt{val}_i\}_{1 \le i \le k})$, we concatenate the attribute name and values of a whole column and serialize it

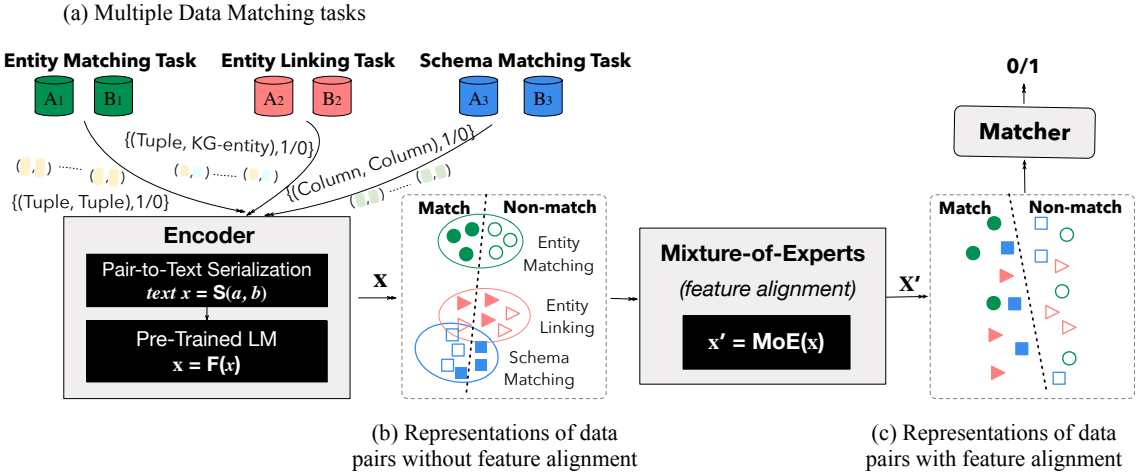


Figure 2: An overview of our proposed `Unicorn` framework. (a) `Unicorn` learns from multiple data matching tasks or datasets. (b) A generic `Encoder` converts any pair of data elements (a, b) into a representation in the form of a high-dimensional vector. (c) A `Mixture-of-Experts` layer aligns matching semantics of multiple tasks by enhancing the learned representation into a better representation. Based on the representation, a `Matcher`, *i.e.*, a binary classifier, decides whether a matches b.

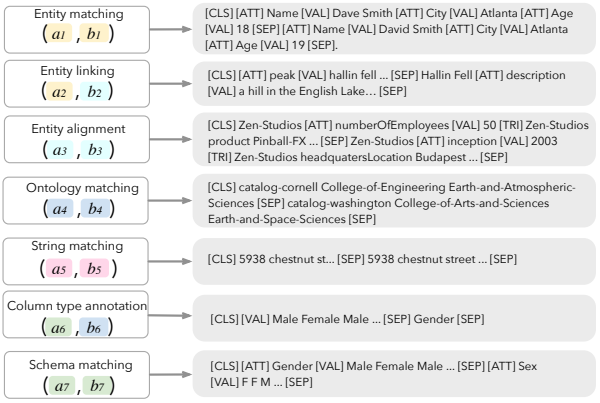


Figure 3: A generic pair-to-text serialization that serializes pairs from data matching tasks into text sequences.

as $\mathsf{S}(\texttt{col}) = \texttt{[ATT] attr [VAL] val}_1\ \texttt{val}_2 \ldots\ \texttt{val}_k$. Note that, in the case of too many values in the column, we randomly select a proportion of the values.

Ontology serialization. Given a tree-based ontology `ont` and a specific $\texttt{node}_k$ in the ontology, we represent it as a sequence by concatenating all nodes in the path from root to $\texttt{node}_k$ as $\mathsf{S}(\texttt{node}_k) = \texttt{node}_1\ \texttt{node}_2 \ldots\ \texttt{node}_k$.

KG-entity serialization. Given a KG-entity, which is also denoted as a subject entity `sub` in the knowledge graph, it has not only some attribute values $\{(\texttt{attr}_i, \texttt{val}_i)\}_{1 \le i \le k}$, but also some relational triples with other entities $\{\langle \texttt{sub}, \texttt{rel}_i, \texttt{obj}_i \rangle\}_{1 \le i \le m}$. Based on the structure, we serialize the KG-entity `sub` into sequence

$$\mathsf{S}(\texttt{sub}) = \texttt{sub [ATT] attr}_1\ \texttt{[VAL] val}_1 \ldots \texttt{[ATT] attr}_k\ \texttt{[VAL] val}_k\ \texttt{[TRI] sub rel}_1\ \texttt{obj}_1 \ldots \texttt{[TRI] sub rel}_m\ \texttt{obj}_m,$$

where `[TRI]` is a special token for specifying relational triples.

Figure 3 depicts the examples of serialized sequences for all data element pairs from the seven matching tasks.

Zero-shot Instruction. For using `Unicorn` in *zero-shot* prediction on new tasks, we further utilize *instruction* [39, 41] to improve the performance. Specifically, we develop a simple *task-agnostic* instruction for data matching tasks in this paper. The basic idea is to define an appropriate natural language template to make downstream matching tasks conforming to the natural language form of pre-training tasks.

3.2.2 *Representation Learning of Serialized Pairs with Pre-trained Language Models*

Representative PLMs include BERT [9], RoBERTa [25], and DeBERTa [19]. A key challenge in `Unicorn` is to support structure-aware encoding, where tokens in serialized sequences carry specific structural information. For example, in entity alignment, the serialized sequence $\mathsf{S}(a_3, b_3)$="`[CLS]` Zen-Studios `[ATT]` numberOfEmployees `[VAL]` 50 `[TRI]` Zen-Studios product Pinball-FX" demonstrates this dependency and importance of token positions.

Conventional self-attention in Transformers may not perform well for structure-aware encoding. Thus, we use DeBERTa [19] for its positional encoding scheme that captures relative token positions. DeBERTa's disentangled attention mechanism, as shown in Equation (3), calculates attention scores considering both contents and relative positions of tokens:

$$\begin{aligned} A_{i,j} &= \{H_i, P_{i|j}\} \times \{H_j, P_{j|i}\}^T \\ &= H_i H_j^T + H_i P_{j|i}^T + P_{i|j} H_j^T + P_{i|j} P_{j|i}^T, \end{aligned} \quad (3)$$

where H_i represents the content of token at the i-th position, and $P_{i|j}$ represents its relative position to the token at the j-th position.

3.3 The Mixture-of-Expert Module

The `Mixture-of-Experts` (MoE) layer's objective in `Unicorn` [35, 16, 22] is to map various task distributions to a shared distribution. By integrating MoE, `Unicorn` supports multiple data matching tasks with diverse semantics and formats, and can be adapted to new tasks.

Formally, $\mathsf{MoE}(\mathbf{x}) : \mathbb{R}^{d_1} \to \mathbb{R}^{d_2}$ transforms an original feature vector $\mathbf{x} \in \mathbb{R}^{d_1}$ to a new vector $\mathbf{x}' \in \mathbb{R}^{d_2}$. The common

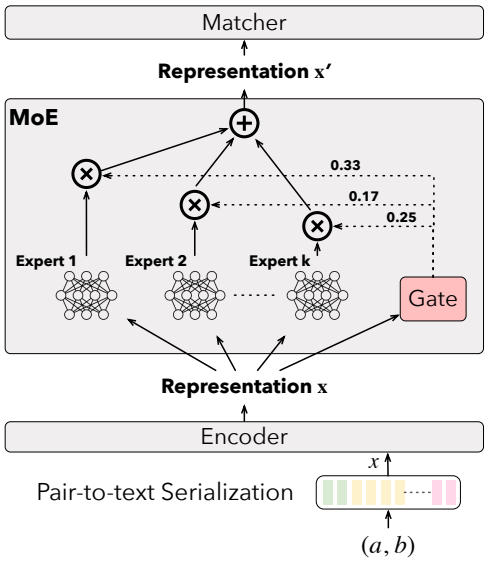


Figure 4: An overview of the `Mixture-of-Experts` layer. Experts are a set of neural networks to map $\mathbf{x}$ to different representations, and Gating is to combine the outputs of Experts according to learned weights based on input $\mathbf{x}$.

`Mixture-of-Experts` architecture [20] consists of Experts and Gating. Experts are neural networks mapping $\mathbf{x}$ to $\mathbf{x}'$, while Gating combines these outputs based on $\mathbf{x}$.

The neural network design and training algorithm for MoE are detailed in Section 3.3.1, and an optimization strategy for expert routing is discussed in Section 3.3.2.

3.3.1 MoE Model Design and Training

Our neural network design for the Experts and Gating in MoE is detailed in Figure 4.

Neural Network Design. Each Expert_i uses a fully-connected layer with `LeakyReLU` activation to map the input feature vector $\mathbf{x}$ to $\mathbf{x}_i$ as shown in Equation (4):

$$\mathbf{x}_i = \texttt{LeakyReLU}(\mathbf{x}W^{(i)}) \tag{4}$$

where $W^{(i)} \in \mathbb{R}^{d_1 \times d_2}$ are trainable parameters.

The Gating component uses two fully-connected layers with `LeakyReLU` and `Softmax` activations to produce a gating vector $\mathbf{g}$, as depicted in Equation (5):

$$\mathbf{g} = \texttt{Softmax}\big(\texttt{LeakyReLU}(\mathbf{x}W_1^{\texttt{G}})W_2^{\texttt{G}}\big) \tag{5}$$

where $W_1^{\texttt{G}} \in \mathbb{R}^{d_1 \times h}$ and $W_2^{\texttt{G}} \in \mathbb{R}^{h \times k}$ are trainable parameters, h is dimension of the hidden layer, and k is the number of experts.

Based on Equations (4) and (5), we obtain k feature vectors, $\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_k$ through the k Experts, and then use weighted average to calculate a unified representation as $\mathbf{x}'$,

$$\mathbf{x}' = \text{MoE}(\mathbf{x}) = g_1 \cdot \mathbf{x}_1 + g_2 \cdot \mathbf{x}_2 + \cdots + g_k \cdot \mathbf{x}_k. \tag{6}$$

End-to-End Model Training. The `Encoder`, MoE, and `Matcher` components of `Unicorn` are trained end-to-end. The output $\mathbf{x}'$ of MoE is input to `Matcher` for predictions $\hat{y} = \text{M}(\mathbf{x}')$. Loss is computed using cross entropy function $\mathcal{L}_{\text{CE}}$ over labeled matching/non-matching pairs.

3.3.2 MoE Optimization for Expert Routing

Training MoE faces the challenge of input vectors $\{\mathbf{x}\}$ favoring a few Experts over others, affecting performance. To tackle this, we introduce an optimization strategy for Expert routing, addressing two objectives: (1) ensuring balanced use of all Experts across the training set, inspired by Sparsely-Gated MoE [35], and (2) assigning specific experts to specific training pairs for better performance.

To achieve balanced utilization, we calculate an Expert utilization vector $\mathbf{u}$ (Equation (7)) and a *load balancing loss* $\mathcal{L}_{\texttt{Bal}}$, which is the squared coefficient of variation of $\mathbf{u}$:

$$\mathbf{u} = (\sum_{\mathbf{x}\in\mathcal{D}} g_1, \sum_{\mathbf{x}\in\mathcal{D}} g_2, \ldots, \sum_{\mathbf{x}\in\mathcal{D}} g_k), \tag{7}$$

where $\sum_{\mathbf{x}\in\mathcal{D}} g_i$ is the sum of gating weights for Expert_i on all the training examples in $\mathcal{D}$.

$$\mathcal{L}_{\texttt{Bal}} = [\frac{\sigma(\mathbf{u})}{\mu(\mathbf{u})}]^2 \tag{8}$$

where $\sigma(\mathbf{u})$ and $\mu(\mathbf{u})$ are respectively standard deviation and mean of the utilization vector $\mathbf{u}$.

For the second objective, we calculate the *entropy* of the gating vector $\mathbf{g}$ for each training example, defining an entropy loss function $\mathcal{L}_{\texttt{Ent}}$:

$$\mathcal{L}_{\texttt{Ent}} = \mathbb{E}_{(\mathbf{x}',y)} - \sum_{i=1}^{k} g_i \cdot \log(g_i) \tag{9}$$

The new loss function for training `Unicorn` is:

$$\mathcal{L}_{\texttt{new}} = \mathcal{L} + \mathcal{L}_{\texttt{Bal}} + \mathcal{L}_{\texttt{Ent}} \tag{10}$$

4. EXPERIMENTS

4.1 Experimental Setup

Dataset & Metrics. Recall that `Unicorn` so far supports seven types of common data matching tasks (see Section 2), including entity matching, entity linking, entity alignment, string matching, column type annotation, schema matching, and ontology matching. We use widely-recognized datasets for evaluation, and the details of datasets and evaluation metrics are listed in Table 1.

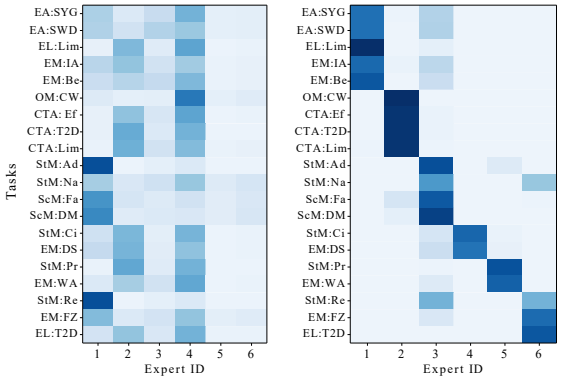
Implementation Details of `Unicorn`. For the PLMs, we use the pre-trained base size checkpoints directly on the Hugging Face. For DeBERTa-base models, they use 12 transformer layers and output a 768 dimensional hidden embedding. For the MoE layer, we choose expert number from 2 to 15, and set hidden dimensions of gate and output size of experts from $\{384, 768, 1024\}$ according to the performance of validation set. A single fully connected layer is used for `Matcher` to output matching probabilities. We choose learning rate from {3e-5, 3e-6}, set batchsize as 32, and use maximum epoch number as 10.

4.2 Evaluation on Unified Prediction

This section presents the experimental results for unified prediction evaluation (see Section 3). We train a shared model `Unicorn` via multi-tasking over all training sets and report performance on various test sets. Due to the space limit, we omit the empirical study of evaluating different PLMs, and only report the result: we find that DeBERTa is the most suitable encoder for `Unicorn`, which requires structure-aware encoding and high generalization.

Table 1: **Dataset Statistics, where $|A|$ (or $|B|$) is the number of data elements in set A (or B) in each task, # Matches (# Non-Matches) is the number of matches (non-matches) in the labeled set $\mathcal{D}$ of the task.**

Task Type	Metric	Task	$\|A\|$	$\|B\|$	# Matches	# Non-Matches
Entity Matching (EM)	F1	Walmart-Amazon (WA)	2,554	22,074	962	9,280
		DBLP-Scholar (DS)	2,616	64,263	5,347	23,360
		Fodors-Zagats (FZ)	533	331	110	836
		iTunes-Amazon (IA)	6,907	55,923	132	407
		Beer (Be)	4,345	3,000	68	382
Column Type Annotation (CTA)	Acc.	Efthymiou (Ef)	620	31	620	18,600
		T2D (T2D)	383	37	383	13,788
		Limaye (Lim)	174	27	179	4,519
Entity Linking (EL)	F1	T2D (T2D)	11,650	26,025	20,666	131,945
		Limaye (Lim)	659	4,166	1,447	36,020
String Matching (StM)	F1	Address (Ad)	24,650	29,531	9,850	1,062
		Names (Na)	10,341	15,396	5,132	2,763
		Researchers (Re)	8,342	43,549	4,556	4,767
		Product (Pr)	2,554	22,074	1,154	79,310
		Citation (Ci)	2,616	64,263	5,347	34,152
Schema Matching (ScM)	F1	Fabricated Datasets (Fa)	11,172	11,352	7,692	109,762
		DeepMDatasets (DM)	41	41	41	268
Ontology Matching (OM)	Acc.	Cornell -Washington (CW)	176	166	53	285
Entity Alignment (EA)	Hits@1	SRPRS: DBP-YG (SYG)	15,000	15,000	15,000	38,891
		SRPRS: DBP-WD (SWD)	15,000	15,000	15,000	38,492



(a) `Unicorn` (b) `Unicorn++`

Figure 5: Visualization of average utilization weights of the Experts in each task. (a) For `Unicorn` with typical MoE, all tasks mainly use the first four Experts, while the weights of the Experts are even. (b) For `Unicorn++` with optimized MoE, the distinction of the Experts is more obvious.

Exp-1: Which strategy performs well in `Mixture-of-Experts` of `Unicorn`?

Table 2 summarizes the results for all tasks. We compare three variants: `Unicorn` w/o MoE (only an `Encoder` and a `Matcher`), standard `Unicorn` (with an `Encoder`, a MoE layer, and a `Matcher`), and `Unicorn++` (an optimized `Unicorn` with MoE for Expert routing, see Section 3.3.2).

The experimental results show `Unicorn` and `Unicorn++` outperform `Unicorn` w/o MoE, highlighting the effectiveness of the MoE layer. Specifically, `Unicorn++` yields the best results across most tasks, demonstrating the benefits of Expert routing. We analyze the Experts' average utilization weights using heatmaps in Figure 5. For `Unicorn`, all tasks predominantly utilize the first four Experts with evenly distributed weights (Figure 5 (a)). In contrast, `Unicorn++` shows a clearer distinction among Experts (Figure 5 (b)), with similar tasks aggregating to specific experts (e.g., EA tasks to Expert 1, CTA tasks to Expert 2). This indicates the efficacy of our optimization, which includes additional loss functions in Equation (10).

Finding 1: The* MoE *intermediate layer is helpful for* `Unicorn`*, while our* MoE *optimization for* Expert *routing further improves the overall performance.

Exp-2: How does a unified model `Unicorn` (*i.e.,* trained using labeled datasets from multiple tasks) compare with specific models (*i.e.,* each model is separately trained for only one task)?

We compare performance with the previous SOTA methods in Table 2, where previous SOTA shows the results of the best task-specific model, which are reported by the existing papers, for each corresponding task.

The experimental results show that our unified models (*i.e.,* `Unicorn` and `Unicorn++`) outperform the previous SOTA methods on 15 over 20 tasks. On average, `Unicorn++` achieves a score of 94.56, compared to the previous SOTA's 91.84. Moreover, another advantage of our approach is the significantly reduced model size due to task unification: 147M for `Unicorn` (comprising 139M for DeBERTa-base and 8M for MoE layer) versus 995.5M for previous SOTA methods, where we compute the number of parameters of models to measure the model size.

The performance superiority of `Unicorn` is attributed to its **multi-task learning** that enables the unified model to learn from multiple tasks and multiple datasets to make full use of knowledge sharing.

Finding 2: Our unified model achieves better performance on most datasets and on average, compared with the SOTA specific models trained for ad-hoc tasks and datasets separately.

4.3 Evaluation on Zero-Shot Prediction

Exp-3: How does Unicorn perform on unseen tasks with a zero-shot setting?

We evaluate the zero-shot capability of our unified model `Unicorn`. For each task type, we randomly choose one task as new **unlabeled** task for testing, and only use the remaining tasks for multi-task training of `Unicorn`. As shown in Table 3, the results of `Unicorn` with **zero** label are comparable with previous SOTA methods with many labels.

Finding 3:* `Unicorn` *can be effectively applied to new tasks in a* zero-shot *setting such that pairs in the new task are unlabeled.

Exp-4: Whether our proposed MoE and instruction techniques are helpful in the zero-shot setting?

Table 2: **The Overall Result for Unified Prediction.** `Unicorn` **w/o** MoE **is a variant of** `Unicorn` **that has no** MoE **layer.** `Unicorn` **is our proposed framework with** `Encoder`**,** MoE **and** `Matcher`**.** `Unicorn++` **is improved with** MoE **optimization for** Expert **routing.**

Type	Task	Metric	Unicorn w/o MoE	Unicorn	Unicorn++	Previous SOTA (Paper)
EM	Walmart-Amazon	F1	85.12	86.89	**86.93**	86.76 (Ditto [24])
	DBLP-Scholar	F1	95.38	95.64	**96.22**	95.6 (Ditto [24])
	Fodors-Zagats	F1	97.78	**100**	97.67	**100** (Ditto [24])
	iTunes-Amazon	F1	94.74	96.43	**98.18**	97.06 (Ditto [24])
	Beer	F1	90.32	90.32	87.5	**94.37** (Ditto [24])
CTA	Efthymiou	Acc.	98.08	98.42	**98.44**	90.4 (TURL [8])
	T2D	Acc.	98.81	99.14	**99.21**	96.6 (HNN+P2Vec [2])
	Limaye	Acc.	96.11	96.75	**97.32**	96.8 (HNN+P2Vec [2])
EL	T2D	F1	79.96	91.96	**92.25**	85 (Hybrid I [15])
	Limaye	F1	83.12	86.78	**87.9**	82 (Hybrid II [15])
StM	Address	F1	97.81	98.68	99.47	**99.91** (Falcon [30])
	Names	F1	86.12	91.19	**96.8**	95.72 (Falcon [30])
	Researchers	F1	96.59	97.66	**97.93**	97.81 (Falcon [30])
	Product	F1	84.61	82.9	**86.06**	67.18 (Falcon [30])
	Citation	F1	96.34	96.27	**96.64**	90.98 (Falcon [30])
ScM	FabricatedDatasets	Recall	81.19	**89.6**	89.35	81 (Valentine [21])
	DeepMDatasets	Recall	66.67	96.3	96.3	**100** (Valentine [21])
OM	Cornell-Washington	Acc.	90.64	**92.34**	90.21	80 (GLUE [11])
EA	SRPRS: DBP-YG	Hits@1	99.46	99.67	99.49	**100** (BERT-INT [36])
	SRPRS: DBP-WD	Hits@1	97.11	97.22	97.28	**99.6** (BERT-INT [36])
AVG			90.8	94.21	**94.56**	91.84
Model Size			139M	147M	147M	995.5M

Table 3: **Zero-shot Performance of** `Unicorn` **(# of labels is the number of labels needed by SOTA methods).** `Unicorn`**-ins is with the instruction technique.**

Type	Task	Metric	Unicorn w/o MoE	Unicorn	Unicorn-ins	SOTA (# of labels)
EM	DS	F1	90.91	95.39	**97.08**	95.6 (22,965)
CTA	Lim	Acc.	96.2	96.59	96.5	**96.8** (80)
EL	Lim	F1	74.16	78.92	**82.8**	82 (-)
StM	Pr	F1	60.71	74.92	**78.76**	67.18 (1,020)
ScM	DM	Recall	74.07	92.59	96.3	**100** (-)
EA	SWD	Hits@1	95.55	97.25	96.17	**99.6** (4,500)
AVG			81.93	89.28	**91.27**	90.2

Table 4: **Zero-shot Performance of** `Unicorn` **for new unseen task types. (# of labels is the number of labels needed by SOTA methods).**

Type	Task	Metric	Unicorn-ins	SOTA (# of labels)
EM	DS	F1	94.5	**95.6** (22,965)
CTA	Lim	Acc.	96.23	**96.8** (80)
EL	Lim	F1	79.59	**82** (-)
StM	Pr	F1	**74.26**	67.18 (1,020)
ScM	DM	Recall	88.89	**100** (-)
EA	SWD	Hits@1	97	**99.6** (4,500)
AVG			88.41	**90.2**

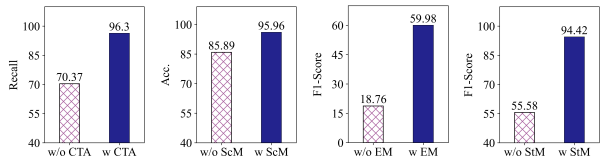
We evaluate the performance of zero-shot learning on new tasks for three `Unicorn` variants: `Unicorn` w/o MoE, `Unicorn`, and `Unicorn`-ins (with zero-shot instruction as presented in Section 3.2.1). The results in Table 3 reveal that `Unicorn` w/o MoE underperforms compared to the other variants, highlighting the MoE Layer's role in leveraging knowledge from seen tasks for new unseen tasks. The MoE layer effectively aligns distributions between new and existing tasks, facilitating adaptation to new tasks.

`Unicorn`-ins shows the best performance, surpassing standard `Unicorn` by approximately 2% on average (Table 3). Recent studies like Wang et al. [39, 41, 28] explore advanced instruction techniques for such tasks, which we aim to investigate in future work.

Finding 4: There is great potential for studying `Mixture-of-Experts` ***and instruction on*** `Unicorn` ***to improve the performance of zero-shot predictions.***

Exp-5: How does `Unicorn` **perform for unseen tasks of totally new task types?**

We evaluate the performance of `Unicorn` for unseen tasks of *totally new task types*. To this end, we conduct experiments with `Unicorn`-ins, a version of `Unicorn` improved by our proposed instruction technique. The results are reported in Table 4, where each row represents an unseen task of a total new task type. Take the first row of Table 4 as an example. For testing DBLP-Scholar of entity matching, we remove all the tasks/datasets of entity matching from the training data of `Unicorn`-ins. We can see that the performance of `Unicorn`-ins is comparable to that of the SOTA specific models, *e.g.,* 88.41 vs. 90.2 on average, and some-



(a) ScM:DM (b) CTA:Lim (c) StM:Pr (d) EM:DS

Figure 6: Knowledge sharing capability of `Unicorn` across different task types. (a) and (b) evaluate knowledge sharing between schema matching (ScM) and column type annotation (CTA), while (c) and (d) consider entity matching (EM) and string matching (StM).

times `Unicorn`-ins is even better. For example, on the Product task, `Unicorn`-ins with zero labels performs better than the SOTA specific model with 1,020 labels (74.26 vs. 67.18). The results show that our unified model is competitive to SOTA specific models even if it has never seen the type of a task and has no specific labels from the task.

The results inspire us to conduct a case study on the *knowledge sharing capability* of `Unicorn` across different task types. We analyze how the performance of one task type is affected when trained *with* or *without* another task type.

(1) Similar Task Types. We explore task types with the same data elements, such as column type annotation (CTA) and schema matching (ScM), both involving the *column* element. Figure 6 (a) indicates that training `Unicorn` with CTA tasks significantly benefits ScM tasks (e.g., DeepMDatasets, 96.3 vs. 70.37 on Recall). Similarly, Figure 6 (b) shows ScM tasks enhance CTA task performance (e.g., Limaye, 95.96 vs. 85.89 on Acc.).

(2) Similar Tasks. We also examine tasks in similar domains, such as products or citations. For instance, entity matching (EM) and string matching (StM) tasks in the product domain show knowledge sharing benefits. Figure 6 (c) shows that training `Unicorn` with EM tasks is helpful for improving the performance of the StM task Product (59.98 vs. 18.76 on F1), and vice versa (see Figure 6 (d)).

Finding 5: `Unicorn` ***performs well for unseen tasks of totally new task types in the zero-shot setting. Moreover,*** `Unicorn` ***enables the possible opportunities of knowledge sharing among different task types.***

5. RELATED WORK

Data Matching Tasks. The "data matching" process is central to most, if not all, data integration problems. Existing approaches (*e.g.,* Ditto [24], TURL [8], BERT-INT [36] and so on) try to solve each problem separately, *e.g.,* using different frameworks or different training methods. Moreover, these solutions are not only task-specific, but also oftentimes dataset-specific (*e.g.,* for each new task. Different from them, `Unicorn` aims at a unified matching model that can be used for multiple matching tasks and datasets.

Transformer-based Language Models. The Transformer architecture, introduced in "Attention Is All You Need" [38], initially aimed at natural language processing, has been successfully adapted for image processing [12] and multimodality tasks like DeepMind's Gato [34]. Furthermore, Transformer-based models such as BERT [9] and RoBERTa [25] have been applied in database tasks, including Text-to-SQL translation (PASTA [18], SCPromt [17]) and data lake querying (Symphony [3]), as well as data integration tasks like entity matching (Ditto [24]) and blocking in entity matching (DeepBlocker [37]). Motivated by these advancements, this paper explores the potential of a unified Transformer-based model for various matching tasks essential in data integration and management.

Mixture of Experts Models. `Mixture-of-Experts` abbreviated as MoE and referenced in [20, 35, 16, 22], is an ensemble technique where multiple experts focus on different sub-tasks. Each expert in MoE specializes in a segment of the task space, with a gating network integrating their outputs. Widely adopted by major tech companies, MoE has shown its effectiveness in diverse applications.

In the context of `Unicorn`, MoE offers significant advantages. Primarily, `Unicorn`, designed to handle various data matching tasks, benefits from the specialized focus of each expert in MoE on distinct data types and semantics. Secondly, MoE enhances the scalability and adaptability of `Unicorn`, allowing for easy extension to novel matching tasks, such as the tuple-to-image matching discussed in Section 6.

6. CONCLUSION AND FUTURE WORK

We have proposed `Unicorn`, a unified model for supporting multiple data matching tasks. So far, `Unicorn` supports seven data matching tasks over five different types of data elements. This unified model can enable **knowledge sharing** by learning from multiple tasks and multiple datasets, and also support **zero-shot prediction** for new tasks with zero labeled pairs. We developed a general framework for `Unicorn` that employs an `Encoder`, a `Mixture-of-Experts` and a `Matcher`. We conducted experiments on 20 datasets of the seven matching tasks. Experimental results show that `Unicorn` is not only comparable or even better than SOTA specific models, but also well performs zero-shot learning on unseen new tasks.

We further discuss the future impact and developments that this research may trigger. Firstly, the optimized MoE and instruction we discussed lead to obvious improvements. Thus, it is worthwhile to explore more optimization techniques in the unified model, *e.g.,* using data augmentation to synthesize new labeled data, to reduce human labeling cost and improve model performance. Secondly, another interesting future work is to extend `Unicorn` to support more modalities (*e.g.,* images), such as whether a picture matches a person (*e.g.,* Michael Jordan) in a knowledge graph. Thirdly, recent progress on large language models (LLMs), such as PaLM [4], ChatGPT and GPT-4 [29], has enabled the possibilities of extending `Unicorn` to support all tasks in data preparation, such as error detection, missing value imputation, etc. Despite some very recent attempts [23, 43], it still calls for a unified system that has strong generalizability to support unseen tasks.

7. ACKNOWLEDGEMENT

This work was partly supported by the NSF of China (62122090, 62072461, U1911203, 62072458, 62232009 and 61925205), National Key Research and Development Program of China (2020YFB2104101), the Beijing Natural Science Foundation (L222006), the Research Funds of Renmin University of China, Huawei, TAL education, and Beijing National Research Center for Information Science and Technology (BNRist).

8. REFERENCES

[1] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[2] J. Chen, E. Jiménez-Ruiz, I. Horrocks, and C. Sutton. Learning semantic annotations for tabular data. *arXiv preprint arXiv:1906.00781*, 2019.

[3] Z. Chen, Z. Gu, L. Cao, J. Fan, S. Madden, and N. Tang. Symphony: Towards natural language query answering over multi-modal data lakes. In *2023 Conference on Innovative Data Systems Research (CIDR)*, 2023.

[4] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann, P. Schuh, K. Shi, S. Tsvyashchenko, J. Maynez, A. Rao, P. Barnes, Y. Tay, N. Shazeer, V. Prabhakaran, E. Reif, N. Du, B. Hutchinson, R. Pope, J. Bradbury, J. Austin, M. Isard, G. Gur-Ari, P. Yin, T. Duke, A. Levskaya, S. Ghemawat, S. Dev, H. Michalewski, X. Garcia, V. Misra, K. Robinson, L. Fedus, D. Zhou, D. Ippolito, D. Luan, H. Lim, B. Zoph, A. Spiridonov, R. Sepassi, D. Dohan, S. Agrawal, M. Omernick, A. M. Dai, T. S. Pillai, M. Pellat, A. Lewkowycz, E. Moreira, R. Child, O. Polozov, K. Lee, Z. Zhou, X. Wang, B. Saeta, M. Diaz, O. Firat, M. Catasta, J. Wei, K. Meier-Hellstern, D. Eck, J. Dean, S. Petrov, and N. Fiedel. Palm: Scaling language modeling with pathways. *J. Mach. Learn. Res.*, 24:240:1–240:113, 2023.

[5] V. Christophides, V. Efthymiou, T. Palpanas, G. Papadakis, and K. Stefanidis. An overview of end-to-end entity resolution for big data. *ACM Computing Surveys (CSUR)*, 53(6):1–42, 2020.

[6] R. Collobert and J. Weston. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th international conference on Machine learning*, pages 160–167, 2008.

[7] D. Deng, R. C. Fernandez, Z. Abedjan, S. Wang, M. Stonebraker, A. K. Elmagarmid, I. F. Ilyas, S. Madden, M. Ouzzani, and N. Tang. The data civilizer system. In *8th Biennial Conference on Innovative Data Systems Research, CIDR 2017, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*, 2017.

[8] X. Deng, H. Sun, A. Lees, Y. Wu, and C. Yu. TURL: table understanding through representation learning. *Proceedings of the VLDB Endowment*, 14(3):307–319, 2020.

[9] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[10] A. Doan, A. Halevy, and Z. Ives. *Principles of data integration*. Elsevier, 2012.

[11] A. Doan, J. Madhavan, R. Dhamankar, P. Domingos, and A. Halevy. Learning to match ontologies on the semantic web. *The VLDB journal*, 12(4):303–319, 2003.

[12] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. In *ICLR*, 2021.

[13] H. L. Dunn. Record linkage. *American Journal of Public Health*, 36(12):1412–1416, 1946.

[14] M. Ebraheem, S. Thirumuruganathan, S. Joty, M. Ouzzani, and N. Tang. Distributed representations of tuples for entity resolution. *Proceedings of the VLDB Endowment*, 11(11):1454–1467, 2018.

[15] V. Efthymiou, O. Hassanzadeh, M. Rodriguez-Muro, and V. Christophides. Matching web tables with knowledge base entities: from entity lookups to entity embeddings. In *International Semantic Web Conference*, pages 260–277. Springer, 2017.

[16] W. Fedus, B. Zoph, and N. Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity, 2022.

[17] Z. Gu, J. Fan, N. Tang, L. Cao, B. Jia, S. Madden, and X. Du. Few-shot text-to-sql translation using structure and content prompt learning. *Proceedings of the ACM on Management of Data*, 1(2):1–28, 2023.

[18] Z. Gu, J. Fan, N. Tang, P. Nakov, X. Zhao, and X. Du. PASTA: table-operations aware fact verification via sentence-table cloze pre-training. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing, EMNLP 2022, Abu Dhabi, United Arab Emirates, December 7-11, 2022*, pages 4971–4983. Association for Computational Linguistics, 2022.

[19] P. He, X. Liu, J. Gao, and W. Chen. Deberta: Decoding-enhanced bert with disentangled attention. *arXiv preprint arXiv:2006.03654*, 2020.

[20] R. A. Jacobs, M. I. Jordan, S. J. Nowlan, and G. E. Hinton. Adaptive mixtures of local experts. *Neural computation*, 3(1):79–87, 1991.

[21] C. Koutras, G. Siachamis, A. Ionescu, K. Psarakis, J. Brons, M. Fragkoulis, C. Lofi, A. Bonifati, and A. Katsifodimos. Valentine: Evaluating matching techniques for dataset discovery. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pages 468–479. IEEE, 2021.

[22] D. Lepikhin, H. Lee, Y. Xu, D. Chen, O. Firat, Y. Huang, M. Krikun, N. Shazeer, and Z. Chen. Gshard: Scaling giant models with conditional computation and automatic sharding. *arXiv preprint arXiv:2006.16668*, 2020.

[23] P. Li, Y. He, D. Yashar, W. Cui, S. Ge, H. Zhang, D. R. Fainman, D. Zhang, and S. Chaudhuri. Table-gpt: Table-tuned GPT for diverse table tasks. *CoRR*, abs/2310.09263, 2023.

[24] Y. Li, J. Li, Y. Suhara, A. Doan, and W. Tan. Deep entity matching with pre-trained language models. *Proceedings of the VLDB Endowment*, 14(1):50–60, 2020.

[25] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

[26] J. Ma, Z. Zhao, X. Yi, J. Chen, L. Hong, and E. H.

Chi. Modeling task relationships in multi-task learning with multi-gate mixture-of-experts. In *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 1930–1939, 2018.

[27] S. Mudgal, H. Li, T. Rekatsinas, A. Doan, Y. Park, G. Krishnan, R. Deep, E. Arcaute, and V. Raghavendra. Deep learning for entity matching: A design space exploration. In *Proceedings of the 2018 International Conference on Management of Data*, pages 19–34, 2018.

[28] A. Narayan, I. Chami, L. J. Orr, and C. Ré. Can foundation models wrangle your data? *CoRR*, abs/2205.09911, 2022.

[29] OpenAI. GPT-4 technical report. *CoRR*, abs/2303.08774, 2023.

[30] G. Paul Suganthan, A. Ardalan, A. Doan, and A. Akella. Smurf: Self-service string matching using random forests. *Proceedings of the VLDB Endowment*, 12(3), 2019.

[31] F. Pourpanah, M. Abdar, Y. Luo, X. Zhou, R. Wang, C. P. Lim, X.-Z. Wang, and Q. J. Wu. A review of generalized zero-shot learning methods. *IEEE transactions on pattern analysis and machine intelligence*, 2022.

[32] Z. Qin, Y. Cheng, Z. Zhao, Z. Chen, D. Metzler, and J. Qin. Multitask mixture of sequential experts for user activity streams. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 3083–3091, 2020.

[33] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *The Journal of Machine Learning Research*, 21(1):5485–5551, 2020.

[34] S. E. Reed, K. Zolna, E. Parisotto, S. G. Colmenarejo, A. Novikov, G. Barth-Maron, M. Gimenez, Y. Sulsky, J. Kay, J. T. Springenberg, T. Eccles, J. Bruce, A. Razavi, A. Edwards, N. Heess, Y. Chen, R. Hadsell, O. Vinyals, M. Bordbar, and N. de Freitas. A generalist agent. *CoRR*, abs/2205.06175, 2022.

[35] N. Shazeer, A. Mirhoseini, K. Maziarz, A. Davis, Q. Le, G. Hinton, and J. Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538*, 2017.

[36] X. Tang, J. Zhang, B. Chen, Y. Yang, H. Chen, and C. Li. Bert-int: a bert-based interaction model for knowledge graph alignment. In *Proceedings of the Twenty-Ninth International Conference on International Joint Conferences on Artificial Intelligence*, pages 3174–3180, 2021.

[37] S. Thirumuruganathan, H. Li, N. Tang, M. Ouzzani, Y. Govind, D. Paulsen, G. Fung, and A. Doan. Deep learning for blocking in entity matching: a design space exploration. *Proceedings of the VLDB Endowment*, 14(11):2459–2472, 2021.

[38] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[39] P. Wang, A. Yang, R. Men, J. Lin, S. Bai, Z. Li, J. Ma, C. Zhou, J. Zhou, and H. Yang. Ofa: Unifying architectures, tasks, and modalities through a simple sequence-to-sequence learning framework. In *International Conference on Machine Learning*, pages 23318–23340. PMLR, 2022.

[40] W. Wang, V. W. Zheng, H. Yu, and C. Miao. A survey of zero-shot learning: Settings, methods, and applications. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 10(2):1–37, 2019.

[41] Y. Wang, S. Mishra, P. Alipoormolabashi, Y. Kordi, A. Mirzaei, A. Naik, A. Ashok, A. S. Dhanasekaran, A. Arunkumar, D. Stap, et al. Super-naturalinstructions: Generalization via declarative instructions on 1600+ nlp tasks. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 5085–5109, 2022.

[42] D. Zhang, Y. Suhara, J. Li, M. Hulsebos, Ç. Demiralp, and W. Tan. Sato: Contextual semantic type detection in tables. *Proceedings of the VLDB Endowment*, 13(11):1835–1848, 2020.

[43] H. Zhang, Y. Dong, C. Xiao, and M. Oyamada. Jellyfish: A large language model for data preprocessing. *CoRR*, abs/2312.01678, 2023.

[44] Y. Zhang and Q. Yang. A survey on multi-task learning. *IEEE Transactions on Knowledge and Data Engineering*, 34(12):5586–5609, 2021.

Technical Perspective: Graph Theory for Data Privacy: A New Approach for Complex Data Flows

Elena Ferrari
Department of Theoretical and Applied Sciece
University of Insubria, Varese (Italy)
elena.ferrari@uninsubria.it

Nearly all of the world's population now uses online services that request personal information, covering almost every aspect of our lives. The abundance of personal data in digital form has brought incredible benefits to end users, enabling them to access personalized and advanced services based on the analysis of the data collected. This capability has dramatically improved the user experience in various application domains, ranging from healthcare to e-commerce, finance, logistics, and entertainment, to name a few. Numerous technological advancements in the field of big data have enabled this massive processing of personal data, and recent advances in AI data processing capabilities will expand the ways in which service providers will use personal data in the coming years. Machine learning algorithms, powered by AI, will be used to make increasingly accurate predictions about user behavior by uncovering hidden correlations within massive data sets. There is therefore a tension between the desire to fully exploit personal data in such ecosystems and the need to provide strong privacy and transparency guarantees to the individuals whose data is being exploited. Privacy protection is further complicated because data processing is typically not performed in isolation but through pipelines of different services, with each step making inferences about the personal data consumed by the services in subsequent steps.

Privacy and transparency in data processing have also been driven by the emergence of privacy regulations, the most notable being the European General Data Protection Regulation (GDPR), which has inspired many subsequent privacy laws, such as the California Consumer Privacy Act (CCPA), or the Brazilian Lei Geral de Proteção de Dados (LGPD). A pillar of the GDPR is that end users should have control over how their data is used, and for what purposes the data is managed and processed.

As a result, the field of privacy-preserving techniques has been a very active research area in the last decades, and many privacy-enhancing techniques have been proposed. However, the majority of these techniques protect privacy by masking or perturbing data, the main notable ones being those based on differential privacy, which is today a gold standard of data privacy. However, how to strike a balance between utility and privacy achieved through data modification is still an open issue in many use-case scenarios.

In contrast, the development of solutions to help end users in controlling the flow of their personal data when data are processed by companies' data processing workflows is still an open research issue. The main challenge is how to be compliant with user privacy constraints by, at the same time maximizing data utility for the service provider. A further relevant dimension of the problem is that these processing workflows often target massive processing of personal information. For instance, the highlighted paper reports that the data processing workflow of Meta contains over 12 million service instances and over 180,000 communication edges between services. Developing a sound and scalable solution for this setting is far from being trivial.

The highlighted paper addresses this challenging scenario by proposing a theoretically sound and effective solution. The key intuition of the paper is the use of a graph model: data processing is modelled as a graph where, in addition to nodes representing the types of data collected and the processing algorithms, there is another class of nodes denoting the purposes of data processing. Purpose nodes are linked to an associated utility. In this way, privacy constraints can be modeled as pairs of nodes in the graph that should be separated in order to satisfy the constraints. The problem of satisfying users' privacy constraints is then expressed as a graph optimization problem: selecting the nodes in the graph to separate in a way that maximizes utility. The authors show that this problem is NP-hard and present several alternative heuristics and related algorithms, precisely characterizing their underlying computational complexity as well as the accuracy achieved.

The authors then provide a comprehensive performance evaluation for a relevant instance of the defined optimization problem targeting linearly additive utility functions. Experiments on different datasets show that the designed algorithms can achieve a near-optimal solution in a few seconds for scenarios with graphs of thousands of nodes and tens of user constraints.

The paper not only presents a solid and practical solution to a relevant and timely problem in the area of privacy-preserving data management, but also opens the way for many follow-up research contributions, some of the most important of which are discussed at the end of the paper.

Graph Theory for Consent Management: A New Approach for Complex Data Flows

Dorota Filipczuk*
Microsoft
Dronning Eufemias gate 71
0194 Oslo, Norway
dorotaf@acm.org

Enrico H. Gerding
University of Southampton
University Road
Southampton SO17 1BJ, UK
eg@ecs.soton.ac.uk

George Konstantinidis
University of Southampton
University Road
Southampton SO17 1BJ, UK
G.Konstantinidis@soton.ac.uk

ABSTRACT

Through legislation and technical advances users gain more control over how their data is processed, and they expect online services to respect their privacy choices and preferences. However, data may be processed for many different purposes by several layers of algorithms that create complex data workflows. To date, there is no existing approach to automatically satisfy fine-grained privacy constraints of a user in a way which optimises the service provider's gains from processing. In this article, we propose a solution to this problem by modelling a data flow as a graph. User constraints and processing purposes are pairs of vertices which need to be disconnected in this graph. We show that, in general, this problem is NP-hard and we propose several heuristics and algorithms. We discuss the optimality versus efficiency of our algorithms and evaluate them using synthetically generated data. On the practical side, our algorithms can provide nearly optimal solutions for tens of constraints and graphs of thousands of nodes, in a few seconds.

1. INTRODUCTION

Personal data processing is at the core of many computing systems. In some complex ones, such as those owned by Netflix, Meta or Amazon, users' data is automatically processed by microservices. Typically, personal data enters the system through front-end applications, which send requests to a subset of services. There, each individual service performs a specific business function independent from the rest of the services, often producing predictions or inferences that are in turn consumed by other services. For instance, in a social media system, the user's location may be used by a service responsible for processing user profile information that then sends this data to a usage analytics service and to a service for suggesting groups to join. Consequently, the inferences made by the usage analytics service may be used by an ads-ranking service and the predicted groups of user's interest may be used by a service recommending products to buy. These services next call other services, creating data flows that span over multiple layers of computation. The ultimate goal of this data processing is to eventually satisfy certain business goals by which the service provider gains utility e.g., the ads ranking service is used to increase revenue through personalised advertisement and the product recommendation service may serve the purpose of collecting commission on product sale.

However, individuals should have control over how their data is used. Particularly, in certain regulatory frameworks such as the GDPR [12], unless there exists another legal basis, user's consent is necessary to legally allow personal data to be processed. By opting out of data processing, e.g., not allowing their location data to be used for personal advertisement or product recommendation, users put constraints on the data flow. Enforcing these constraints affects the utility of the service provider who would like the utility loss minimised. What complicates the task even more is its large scale: in modern computing systems, there may be many stages of data processing, in a data flow that involves hundreds of nodes. In fact, Meta's microservice topology contains over 12 million service instances and over 180,000 communication edges between services [15]. In such cases, stopping the data flow to the ads-ranking service or the product recommendation service may affect the quality of the inferences that other services use, and thus, the utility of the service provider.

In this article, we propose a novel approach to finding optimal ways of satisfying privacy constraints automatically while minimising the utility loss for the service provider. We model the data flow as a graph and privacy constraints as pairs of vertices in the graph. Our problem definition is generic, allowing the utility of a purpose node to be a black-box function of the subgraph connected to that node. We formulate the problem as an optimisation problem, where pairs of graph vertices must be disconnected such that utility is maximised (Section 2) and demonstrate it on an example use case (Section 3). We then present a natural instantiation of the generic problem where the utility functions are linearly additive (Section 4).

We present five generic heuristics for implementing the privacy constraints into data workflows (Section 5) with an optimal or approximately optimal global utility, depending on the actual utility function used. We further analyse the (non-)optimality of our heuristics in Section 6. For our experiments (Section 7) we implement the aforementioned linearly additive instance of the problem. Notably, we show that, although computing the optimal solution can be very time-consuming, the proposed heuristics can provide good and efficient approximation alternatives. Our ap-

 DOI: http://dx.doi.org/10.48786/edbt.2023.61.

*This research was conducted while the author was at the University of Southampton.

proach opens a range of new problems that we discuss in Section 8. Finally we juxtapose our approach with related work in Section 9.

2. PROBLEM FORMULATION

We focus on what we call the *Consented Data Workflow* problem (CDW): given user constraints expressed in terms of the vertices that they do not wish to be connected, find a subgraph of the original workflow where these constraints are satisfied. In the face of alternative solutions, the optimal one should minimise the utility loss for the service provider from applying the constraints. In other words, we are looking for the utility-maximising solution subject to the users' privacy constraints.

Formally, our data processing model is a directed graph $G = (V, E)$ with a set of edges E representing the data flow and a set of vertices V representing the stages of data processing. We distinguish three kinds of vertices, i.e. $V = V^U \cup V^A \cup V^P$, where V^U is a set of user data vertices, which represent the types of data collected directly from the user, V^A is a set of algorithm vertices, which represent data processing algorithms that take one or more data types as input, and V^P is a set of purpose vertices, which represent the end goals of data processing.

Furthermore, satisfying the given purposes is what brings service providers utility. For any vertex in V^P, we will have an associated utility that reflects the value processing that this purpose brings to service providers. In practice, the service providers' valuation depends on factors such as the accuracy of the datasets used as the input to the processing algorithms [20, 14]. In particular, where data processing is a multi-stage process, the utility is affected by all stages and all datasets processed for the purpose.

Therefore, in order to calculate the utility of data processing for a given purpose, in our model we look for all vertices and edges that carry the data workflow to the given purpose vertex. Formally, we say that a vertex $v_i \in V$ is *reachable* from a vertex $v_j \in V$ if there exists a path in G, $\{(v_1, v_2),(v_2, v_3),\dots\ (v_{k-1}, v_k)\}$ such that $v_1 = v_j$ and $v_k = v_i$. For each purpose vertex $p \in V^P$ in our graph G, the *reachability subgraph* of p is the graph $G_p = (V_p, E_p)$ where $V_p \subseteq V$ is the set of the vertices that p is reachable from and $E_p \subseteq E$ is the set of edges from G that connect them. If an edge is removed from G, the reachability subgraph of one or more purpose vertices is affected. In general, we write $\mathcal{R}(G_p)$ to denote the set of all subgraphs of the reachability subgraph of p in G - that are still reachability graphs when some edges are removed. Then, to calculate the utility of fulfilling a purpose, for each purpose vertex $p \in V^P$ we define a utility function $u_p : \mathcal{R}(G_p) \to \mathbb{R}_0^+$, which is a function of a reachability subgraph of p.

While u_p can be an arbitrary function dependent on the valuation (or more general, the importance) of datasets in the corresponding reachability subgraph, the valuations of some datasets may influence the valuations of others. To describe the relationships between these valuations in our model, we define a valuation function $\pi : E \to \mathbb{R}_0^+$, representing the valuation of the data propagating through the edge in the data processing system. As we later describe in Section 4, given the reachability subgraph $G_p = (V_p, E_p)$ of a vertex $p \in V^P$, the utility function $u_p(G_p)$ at p can be defined as a function of the valuations of edges in E_p.

In our setting, a user constraint denotes their choice to opt out their data, represented by a user vertex in V^U, from being used for a purpose vertex in V^P. Formally, our set of constraints is a set $\mathcal{N} = \{(v_s, v_t) \mid v_s \in V^U, v_t \in V^P\}$. In order to satisfy the constraints, the initial graph G needs to be modified by removing one or more edges that belong to the paths between pairs (v_s, v_t), such that the utilities u_p are maximised. In essence, our problem is a multi-objective optimisation problem, where the objectives are to maximise u_p, for all $p \in V^P$. The most common approach to multi-objective optimization is to turn the problem into a single-objective optimization using a weighted sum [21]. This allows us to define the utility of G as:

$$U(G) = \sum_{p \in V^P} w_p u_p(G_p), \tag{1}$$

where w_p is the weight of the purpose corresponding to vertex p and G_p is the reachability subgraph of p. Therefore, given $\mathcal{N}$, the CDW problem is to find the consented subgraph of G:

$$G^* = arg \max_{G'} U(G') \tag{2}$$

where $G' = (V, E')$ is a subgraph of G, $E' \subseteq E$, and there is no path from s to t for each $(s, t) \in \mathcal{N}$.

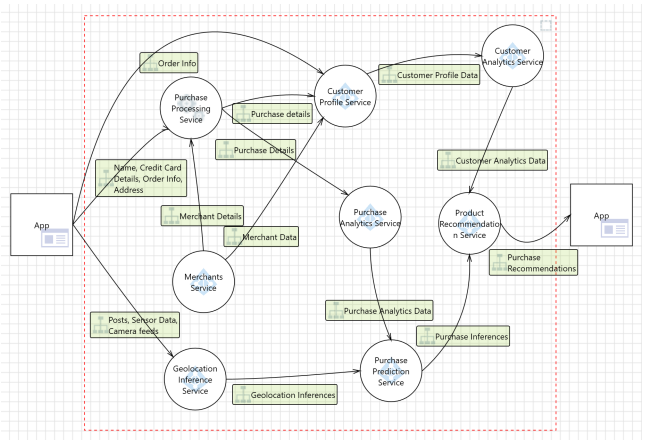


Figure 1: A DFD created with Microsoft Threat Modelling Tool for an example product recommendation feature.

3. RECOMMEND PRODUCT DATA FLOW

Data Flow Diagrams (DFD) [22] are commonly used in the industry to represent data flows in complex systems. For example, Fig. 1 illustrates an example DFD for a single feature: product recommendations. In order to provide product recommendations, user data enters the system though a client application. More specifically, the user's name, credit card details, address and order information are sent to the purchase processing service API. In addition, the order information is sent to the service responsible for constructing customer profiles. Information related to the user's activity such as posts, sensor data or camera feeds are sent to a service producing geolocation-based inferences.

To construct a graph G for this example, firstly, we create vertices V^U for data types collected from the client application: user's name, credit card details, address, order infor-

mation, posts, sensor data and camera feeds. Secondly, we create vertices V^A for all microservices processing the data: the purchase processing service, customer profile service, geolocation inference service, etc. Then, we create a set V^P containing one vertex for the purpose of this feature: serving product recommendations. Finally, we create the edges E connecting vertices in V^U with vertices in V^A the based on the data flow, vertices in V^A with other ones in V^A, and the vertex in V^A representing the service computing product recommendations with the sole purpose vertex in V^P.

An example constraint on data flow of this feature could be a user disagreeing to have their order information used for serving product recommendations. Formally, we would then have a set of constraints $\mathcal{N} = \{(v_s, v_t) \mid v_s \in V^U, v_t \in V^P\}$ where v_s is the vertex corresponding to order information and v_t is our purpose vertex. To solve the problem, we would need to break the data flow from v_s to v_t such that the utility gained from serving product recommendations is maximised. While this example demonstrates a simple use case, in general, a data processing system may consist of many more features on which a user may impose many more constraints.

4. ADDITIVE MODEL

As we prove in [13], in general, CDW is an $\mathcal{NP}$-hard problem. This is because in practice the valuations and utilities defined in Section 2 can be arbitrarily complex functions. In this section, we define a simple but practical instance of the problem, where these functions are linearly additive. In particular, we assume that the valuation function of the data going out of a vertex is linearly additive with respect to the importance of the data on the incoming edges. That is, the value of every node's output is the sum of the values of its inputs. The linear additive model is the simplest model that captures the intuition of each input having an "added value" on the subsequent algorithm. This model may not apply to all settings and, e.g., a sub-additive model might be more appropriate. In many cases the linear additive model could be a reasonable approximation of a model where the inter-dependencies are difficult to measure. Additionally, if we initialise each input edge to have an original value of 1, then the utility of a purpose node, in the this specific setting, sums up how many times the inputs "have been used" in algorithms before reaching the purpose node, thus giving some hint about the overall importance of an input.

In this instance of the problem we consider a DAG $G = (V, E)$ and constraints $\mathcal{N}$, and for each edge $e = (v, v') \in E$, the valuation is defined recursively as follows:

$$\pi(e) = \sum_{e' \in in(v)} \pi(e'). \tag{3}$$

Similarly, we model the utility gained from processing the data for a purpose as a linearly additive function of the data valuations on the incoming edges. That is, for each reachability subgraph G_p, and purpose vertex $p \in V_P$, we define a utility function as follows:

$$u_p(G_p) = \sum_{e \in in(p)} \pi(e). \tag{4}$$

Subsequently we present concrete heuristic implementations for the linear additive case.

5. ALGORITHMS

In this section we focus on a range of algorithms our linearly additive problem. Although there might exist multiple optimal solutions, we design our algorithms looking for a single solution $G^* = (V^*, E^*)$. While some of them offer optimal solutions, others serve as viable heuristics. Note that, even though the algorithms may not be optimal (i.e., utility maximising), all five algorithms always return a *feasible* solution, which is a subgraph of G with no path between each $(s_i, t_i) \in \mathcal{N}$ for $i \in \{1, \ldots, |\mathcal{N}|\}$.

Firstly, a simple heuristic for finding a feasible solution is an algorithm that removes a random edge from each of the paths connecting $(s, t) \in \mathcal{N}$: algorithm RemoveRandomEdge finds all paths from s to t and from each of the paths selects a random edge to remove; then, before the edge is removed, the other edges whose valuation depends on the presence of the given edge in the graph must be updated. In particular, if the valuation of an edge after the update is 0, such edge must also be removed. This solution has a high variance but its run time is polynomial.

Secondly, as the valuation function of the edges is additive, and because the valuation of the incoming edge of an algorithm vertex is always greater or equal than the outgoing one, the removal of the first edge of each path from s to t can serve as another trivial heuristic. Specifically, algorithm RemoveFirstEdge is very similar to RemoveRandomEdge, except that, instead of selecting a random edge, it removes the first edge from each path). This algorithm reflects an approach whereby the user's data is removed entirely and not even collected by the system. Similarly to RemoveRandomEdge, the runtime of this algorithm is polynomial.

Next, we propose a greedy algorithm running in polynomial time. This algorithm follows the heuristic of making locally optimal choices for each constraint. To do so, it uses a polynomial-time algorithm solving the Minimum Cut problem (MinCut) [10, 11] – which is equivalent to the Minimum Multicut when we only have a single constraint $(s, t) \in \mathcal{N}$. Our greedy algorithm RemoveMinCuts, seen in Algorithm 1, first initialises the weights $w(e) = \pi(e) \sum_{p \in r(v)} w_p$ for all edges $e \in E$ (in lines 1 - 4), and then repeatedly calls the MinCut to find the minimum cut that solves MinCut for vertices s and t in $\mathcal{N}$ with weights w. For each edge in the minimum cut, it uses the updateDependencies function to update the valuations of the consecutive edges before removing the given edge. Given that MinCut is known to be solvable in polynomial time [10], the outcome of this heuristic can also be found in polynomial time.

Algorithm 1 RemoveMinCuts

Input: A graph $G = (V, E)$ and a set of constraints $\mathcal{N}$.
Output: A graph G.

1: $w \leftarrow \emptyset$
2: **for all** $e \in E$ **do**
3: $\quad w(e) \leftarrow \pi(e) \sum_{p \in r(v)} w_p$
4: **for all** $(s, t) \in \mathcal{N}$ **do**
5: $\quad$ **for all** $e \in$ MinCut(G, w, s, t) **do**
6: $\quad\quad$ **if** `hasEdge`(G, e) **then**
7: $\quad\quad\quad$ `updateDependencies`(G, e)
8: $\quad\quad\quad$ `removeEdge`(G, e)

Another way of approximating the solution is by converting our problem to MinMC. That is, we can solve MinMC with weights $w(e) = \pi(e) \sum_{p \in r(v)} w_p$ for all edges $e \in E$ and then use the MinMC solution to find a solution. In the same way as RemoveMinCuts, shown in Algorithm 2, RemoveMinMC starts from initialising the weights w. Then, it finds the minimum multicut of graph G for constraints $\mathcal{N}$. Subsequently, for each edge in the minimum multicut, it uses the updateDependencies function to update the valuations of the consecutive edges (in line 8) before removing the given edge.

Algorithm 2 RemoveMinMC

Input: A graph $G = (V, E)$, a set of constraints $\mathcal{N}$.
Output: A graph G.

```
1: w ← ∅
2: for all e ∈ E do
3:     w(e) ← π(e) Σ_{p∈r(v)} w_p
4: multicut ← MinMC(G, N, w)
5: for all e ∈ multicut do
6:     if hasEdge(G, e) then
7:         updateDependencies(G, e)
8:         removeEdge(G, e)
```

Finally, we propose an algorithm that can guarantee achieving an optimal solution. In Algorithm 3, BruteForce is an exhaustive search algorithm that enumerates all feasible candidates for the solution and compares them to eventually output the one that maximises the utility. More specifically, BruteForce starts from finding the set of all paths $\mathcal{A}$ from s to t for all $(s, t) \in \mathcal{N}$, which need to be broken. In order to list all feasible multicuts of G for the given $\mathcal{N}$, the Cartesian product of $\mathcal{A}$ is computed (in line 5). Then, the algorithm systematically checks the utility of G after the removal of each multicut. Importantly, at the beginning of the multicut check, copies are made of the valuation values π' of each edge and the number of paths p' the edge belongs to in G, as well as of the graph G itself. Before an edge of the feasible multicut is removed from the copy of G, the valuation π' and the number of paths p' in the copy of G are updated for its dependencies. At the end of the multicut check, the utility of the copy of G is compared to the utility of the most optimal solution found so far. This way the algorithm can guarantee finding the optimal solution but is exponential even in the best case.

Notably, all of the presented algorithms generalize beyond the linearly additive model. What is specific to this particular model in RemoveMinCuts, RemoveMinMC and BruteForce is the way the weights $w(e)$ are assigned and the dependencies are updated, which depends on the valuation function $\pi(e)$ and weights w_p.

6. OPTIMALITY OF SOLUTIONS

Out of the five algorithms, only BruteForce guarantees an optimal solution. In contrast, it is clear that RemoveRandomEdge does not guarantee an optimal solution – we use it as a benchmark for our evaluation. Let us analyse the properties of the solutions returned by our three remaining heuristics and prove that none of them can guarantee finding an optimal solution even for the linear setting.

Algorithm 3 BruteForce

Input: A graph $G = (V, E)$ and a set of constraints $\mathcal{N}$.
Output: A graph G^*.

```
1: A ← ∅
2: for all (s, t) ∈ N do
3:     A ← A ∪ getAllEdgePaths(G, s, t)
4: multicuts ← cartesianProduct(A)
5: maxUtility ← 0
6: G* ← G
7: for all multicut ∈ multicuts do
8:     G' ← G
9:     π', p ← ∅, ∅
10:    for all e ∈ E do
11:        π'(e) ← π(e)
12:        p(e) ← Σ_{p∈r(v)} w_p
13:    for all e ∈ multicut do
14:        if hasEdge(G', e) then
15:            updateDependencies(G', e, π', p)
16:            removeEdge(G', e)
17:    utility ← U(G')
18:    if utility > maxUtility then
19:        maxUtility ← utility
20:        G* ← G'
21: return G*
```

Firstly, we show that a simple removal of the first edge of each path by the RemoveFirstEdge does not guarantee an optimal solution, i.e., for each $(s_i, t_i) \in \mathcal{N}$, there is at least one path $P = (V_P, E_P) \in \mathcal{A}$ of the form $V_P = \{v_1, v_2, \ldots, v_k\}$, $E_P = \{(v_1, v_2), (v_2, v_3), \ldots, (v_{k-1}, v_k)\}$ where $v_1 = s_i$ and $v_k = t_i$. From each such path $P \in \mathcal{A}$, we could remove edge (v_1, v_2). We refer to (v_1, v_2) as the *first edge*. Let G be a data processing model where $V^U = \{v_1\}$, $V^A = \{v_2\}$, $V^P = \{v_3, v_4\}$, $E = \{(v_1, v_2), (v_2, v_3), (v_2, v_4)\}$ and for each $p \in V^P$, $w_p = 1$. In addition, assume that for edge $e_1 = (v_1, v_2)$, $\pi(e_1) = a$ where $a \in \mathbb{R}_0^+$, and that $\mathcal{N} = \{(v_1, v_3)\}$.

In such case, we use Eq. 3 to calculate the valuation of edges $e_2 = (v_2, v_3)$ and $e_3 = (v_2, v_4)$, which is $\pi(e_2) = \pi(e_3) = a$. We also use Eq. 1 to calculate the initial utility of G, which is $U(G) = 2a$. Given that $\mathcal{N} = \{(v_1, v_3)\}$, we establish that there is one path that needs to be disconnected in order to satisfy the constraints, i.e. $\mathcal{A} = \{P\}$ where $P = (\{v_1, v_2, v_3\}, \{(v_1, v_2), (v_2, v_3)\})$.

Then, we remove the first edge (v_1, v_2) from P. The utility of the resulting graph G_1' is $U(G_1') = 0$, since purpose vertices v_3 and v_4 are now not linked to any user vertex. However, if we instead removed the edge (v_2, v_3), vertex v_4 would still be linked to v_1 and therefore the utility of the resulting graph G_2' would be $U(G_2') = a$. Thus, the removal of the first edge does not provide us with an optimal solution.

Furthermore, RemoveMinCuts does not lead to an optimal solution. Consider a series of graphs $G^0, G^1, \ldots, G^{|\mathcal{N}|}$. These graphs are computed recursively such that $G^0 = G = (V, E)$ and for all $i \in \{1, \ldots, |\mathcal{N}|\}$, $G^i = (V, E^i)$ where $E^i = E^{i-1} \setminus \text{MinCut}(G^{i-1}, s_i, t_i, w)$ corresponds to i-th pair of vertices in $\mathcal{N}$. Given a solution to MinCut, one could transform the problem into a repeated MinCut problem looking for $G^{|\mathcal{N}|}$. While this approach leads to a feasible so-

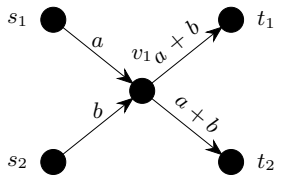


Figure 2: A data processing model where $V^U = \{s_1, s_2\}$, $V^A = \{v_1\}$, $V^P = \{t_1, t_2\}$ and $E = \{(s_1, v_1), (s_2, v_1), (v_1, t_1), (v_1, t_2)\}$.

lution, we show that $G^{|\mathcal{N}|}$ is not an optimal solution. Let G be a graph where $V^U = \{s_1, s_2\}$, $V^A = \{v_1\}$, $V^P = \{t_1, t_2\}$, $E = \{(s_1, v_1), (s_2, v_1), (v_1, t_1), (v_1, t_2)\}$ and for each $p \in V^P$, $w_p = 1$. Assume that for $e_1 = (s_1, v_1)$, $\pi(e_1) = a$ and for $e_2 = (s_2, v_1)$, $\pi(e_2) = b$, where $a, b \in \mathbb{R}_0^+$ and $a > b$ (see Fig. 2). In addition, $\mathcal{N} = \{(s_1, t_1), (s_1, t_2)\}$. We look for $G^2 = (V, E \setminus \textsc{MinCut}(G^1, s_1, t_1, w))$. Thus, we first calculate $G^1 = (V, E \setminus \textsc{MinCut}(G, s_1, t_1, w))$. We observe that there is one path $P = (\{s_1, v_1, t_1\}, \{(s_1, v_1), (v_1, t_1)\})$ between vertices s_1 and t_1. Because $a > b$, $w((v_1, t_1)) = a + b < w((s_1, v_1)) = 2a$. Thus, $G^1 = (V, E \setminus \{(v_1, t_1)\})$. With this information, we return to looking for G^2. We observe that there is one path $P = (\{s_1, v_1, t_2\}, \{(s_1, v_1), (v_1, t_2)\})$ between vertices s_1 and t_2. However, now $w((s_1, v_1)) = a$ and $w((v_1, t_2)) = a + b$. So, $w((s_1, v_1)) < w((v_1, t_2))$ and $G^2 = (V, E \setminus \{(s_1, v_1), (v_1, t_1)\})$. After that, we calculate $U(G^2) = b$. However, we can see that to obtain an optimal solution, it is sufficient to remove (s_1, v_1) only: the optimal solution is $G^* = (V, E \setminus \{(s_1, v_1)\})$, because its utility is $U(G^*) = 2b$. Thus, G^2 is not an optimal solution.

Intuitively, it is reasonable to assume that the optimal solution requires removing no more than one edge per path between s and t for each $(s, t) \in \mathcal{N}$. Let $T \subseteq V^P$ be a set of purpose vertices such that for all $(s, t) \in \mathcal{N}$, $t \in T$. If for each path $P = (V_P, E_P) \in \mathcal{A}$ there is exactly one edge $e \in E_P$ such that $e \in E_{MinMC}$, then the removal of E_{MinMC} reduces the utility of a purpose vertex $t \in T$ by $\sum_{e \in E_t} \pi(e)$ where $E_t \subseteq E_{MinMC}$ is a set of those edges in E_{MinMC} that are within the reachability subgraph of t, G_t. Thus, if the resulting graph after the removal of set E_{MinMC} from G is $G' = (V, E')$, then using Eq. 1, the total loss of utility is:

$$U(G) - U(G') = \sum_{t \in T} \sum_{e \in E_t} w_t \pi(e). \tag{5}$$

This is equivalent to the following equation:

$$U(G') = U(G) - \sum_{e \in E \setminus E'} \pi(e) \sum_{t \in T} w_t. \tag{6}$$

In fact, if we plug Eq. 6 into Equation 2, we have:

$$G^* = arg \max_{G'} \{U(G) - \sum_{e \in E \setminus E'} \pi(e) \sum_{t \in T} w_t\}. \tag{7}$$

Equivalently, we are looking for a subgraph where:

$$E^* = E \setminus \{arg \min_{E \setminus E'} \sum_{e \in E \setminus E'} \pi(e) \sum_{t \in T} w_t\}. \tag{8}$$

Thus, E^* is the set difference of E and the minimum multicut of G given $\mathcal{N}$, where the edge weight is $w(e) = \pi(e) \sum_{t \in T} w_t$. In more detail, the minimum multicut with edge weights $w(e) = \pi(e) \sum_{t \in T} w_t$ can be expressed as:

$$E_{MinMC} = arg \min_{E'} \sum_{e \in E'} \pi(e) \sum_{t \in T} w_t. \tag{9}$$

If we plug Eq. 8 into Eq. 9, then what we are looking for is $G^* = (V, E^*)$ where:

$$E^* = E \setminus E_{MinMC}. \tag{10}$$

Since E_{MinMC} is a solution to MinMC, there is $G^* = (V, E \setminus E_{MinMC})$.

However, removing a single edge from each path does not always result in an optimal solution. Consider a graph G where $V^U = \{s_1, s_2\}$, $V^A = \{v_1\}$, $V^P = \{t_1, t_2\}$, $E = \{(s_1, v_1), (s_2, v_1), (v_1, t_1), (v_1, t_2)\}$ and for each $p \in V^P$, $w_p = 1$. In addition, assume that for $e_1 = (s_1, v_1)$, $\pi(e_1) = a$ and for $e_2 = (s_2, v_1)$, $\pi(e_2) = b$, where $a, b \in \mathbb{R}_0^+$ and $a > b$ (see Fig. 2). Now, let the set of constraints be as follows: $\mathcal{N} = \{(s_1, t_1), (s_1, t_2), (s_2, t_1)\}$. We can see that the optimal solution in this case is $G^* = (V, \{(s_2, v_1), (v_1, t_2)\})$. However, since $(s_1, t_1) \in \mathcal{N}$ and there is a path from s_1 to t_1 in the original graph G, we can observe that the optimal solution G^* does not contain two edges (s_1, v_1) and (v_1, t_2) from that path.Therefore, in general, it is not true that RemoveMinMC can guarantee finding an optimal solution.

7. EXPERIMENTAL EVALUATION

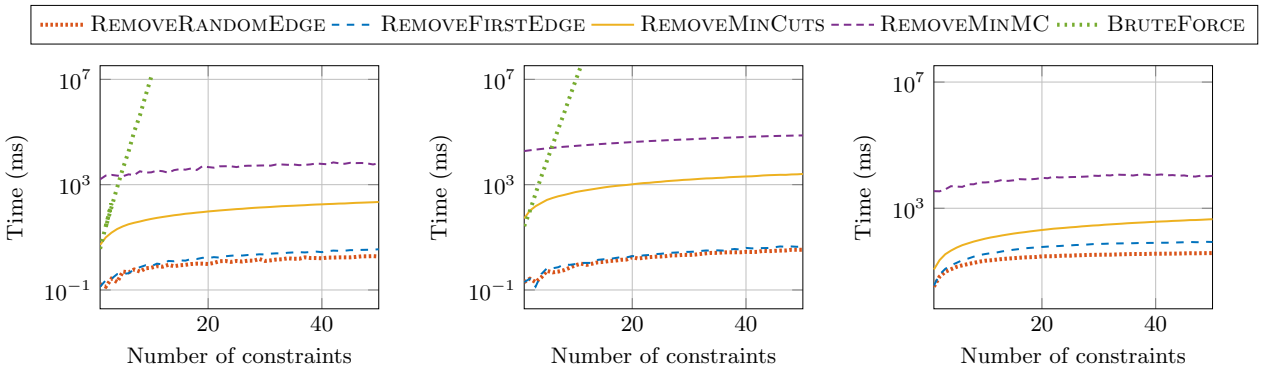
To evaluate the algorithms' performance empirically, we perform experiments on the University of Southampton High Performance Computing service *Iridis 4*, measuring their performance on synthetic data. Our graph generation method includes the following parameters:

- number of constraints $|\mathcal{N}|$;
- number of vertices $|V|$;
- path length k – for any $(s, t) \in \mathcal{N}$, if there is a path $P = \{(v_1, v_2) \dots (v_{k-1}, v_k))\}$ such that $v_1 = s$ and $v_k = t$, then k defines the number of workflow stages, i.e. data that 'flows' from s to t though $k-2$ algorithm nodes;
- vertex distribution vector X_k – proportions of vertices at different data flow stages, e.g. a setting $X_k = (50\%, 25\%, 10\%, 10\%, 5\%)$ represents a scenario for $k = 5$ where half of the vertices are the user data vertices, 45% are the algorithm vertices and 5% are the number of the purpose vertices (NU - non-uniform distribution, U - uniform distribution);
- minimum density d – the proportion of initially generated edges between any two workflow stages.

We prepare three datasets with different configurations of the above parameters, specified in Tab. 1.

We measure the runtime of the algorithms on 100-vertex graphs (dataset 1a), on 10 times larger, 1000-vertex graphs (dataset 1b) and on 100-vertex graphs where the number of edges between each level of data processing is at least 20% of all possible edges (dataset 1c). We observe that the runtime

The Iridis Compute Cluster, `https://cmg.soton.ac.uk/iridis`.



(a) 100-vertex graphs (dataset 1a). (b) 1000-vertex graphs (dataset 1b). (c) Dense graphs (dataset 1c).

Figure 3: The number of constraints vs. the runtime of the algorithms in graphs from dataset 1.

Table 1: Parameter configurations for datasets 1, 2 and 3.

	Dataset 1			**Dataset 2**	**Dataset 3**
	a	**b**	**c**		
$\lvert\mathcal{N}\rvert$	1 – 50	1 – 50	1 – 50	10	5
$\lvert V\rvert$	100	1000	100	150 – 5000	100 – 10000
k	5	5	5	3 – 50	5
X_k	NU	NU	U	U	NU
d	0	0	20%	0	0

Table 2: Comparison of the graph's utility after applying RemoveMinMC and BruteForce.

Number of constraints	RemoveMinMC		BruteForce	
	% of original	SE	% of original	SE
1	97.79	0.32	97.79	0.32
2	95.08	0.35	95.08	0.35
3	92.71	0.58	92.71	0.58
4	90.62	0.57	90.63	0.57
5	88.59	0.75	88.65	0.75
6	86.59	0.72	86.66	0.72
7	84.71	0.72	84.77	0.71
8	83.22	0.70	83.28	0.70
9	81.24	0.69	81.33	0.69
10	79.30	0.69	79.39	0.68

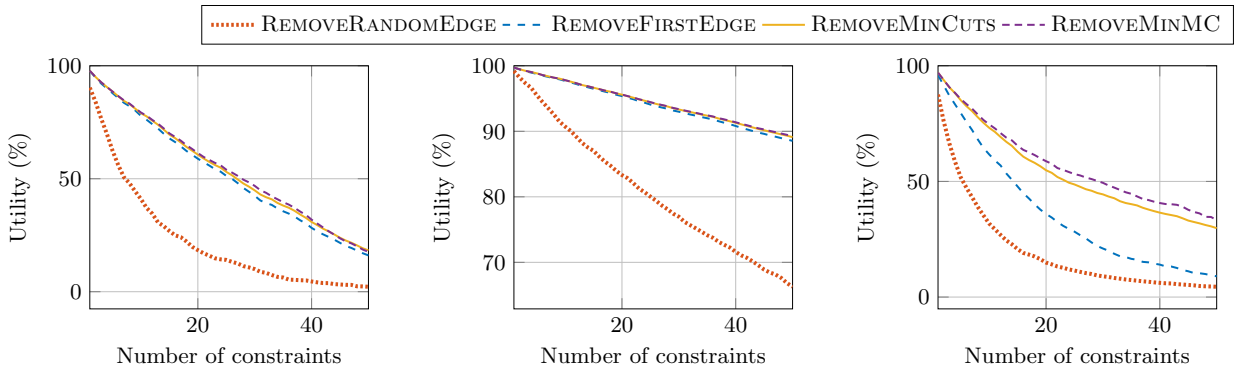
of BruteForce increases rapidly with the increasing number of constraints, reaching an average time of 14838508.46 ms (i.e. over 4 h) for just 10 constraints on dataset 1a and 8563968 ms (i.e. over 2 h) on dataset 1b. For dataset 1c, in most cases, BruteForce is unable to return a result in 60 hours even for just a single pair of constraints. Thus, although BruteForce guarantees finding an optimal solution, its average runtime makes this algorithm impractical. At the same time, the solver-based RemoveMinMC can reach an approximate solution for even 50 constraints on average in 6.51s on dataset 1a, 74.1s on dataset 1b and 11s on dataset 1c. RemoveMinCuts can on average find an approximate solution for 50 constraints in 220.2 ms on dataset 1a, 2.55s on dataset 1b and 450.57 ms on dataset 1c.

Moreover, we observe a change in the graph's utility as the number of constraints grows. Although the exponential runtime of the BruteForce algorithm means we cannot run the experiments for more than 10 constraints, we still compare the results to RemoveMinMC for this limited setting. Results are presented in Tab.2 and show that the utility using RemoveMinMC is nearly optimal in this case. Although the algorithm is only guaranteed to be optimal for specific settings, this empirical outcome suggests that, for graphs with a relatively small number of constraints, RemoveMinMC is likely to provide very accurate solutions. Moreover, Fig. 4, shows that the differences in utility between algorithms is more evident when the graphs are denser, resulting in significantly poorer performance of RemoveMinCuts, RemoveFirstEdge and RemoveRandomEdge. Nonetheless, RemoveMinMC provides the best solution for all three types of graphs.

Next, we observe how the execution time depends on the number of paths between pairs of vertices that connect the constraints. Fig. 5 presents a scatter plot of the runtime of the algorithms and distribution of utility for dataset 1c. In particular, we can see that, in case of RemoveMinCuts and RemoveMinMC, the runtimes increase almost linearly with respect to the number of paths. However, the execution times for these two algorithms differ significantly. For example, for a graph where 822 paths are required to be broken, RemoveMinMC takes 12116 ms to return a solution, whereas RemoveMinCuts can provide one in only 472 ms. Similarly, we can see that the utility decreases as the number of paths connecting constraints increases. Yet again, the utility after executing RemoveMinMC tends to decrease the slowest, reaching on average the utility of 32.27% of the original utility for the graph with 822 paths to be broken. For the same graph, the next best solution is RemoveMinCuts with an accuracy of 24.58%. For comparison, RemoveFirstEdge achieves on average utility of 15.73%.

Next, we apply the algorithms to graphs with a constant number of paths. Since only edges connected to purpose vertices affect the utility, increasing the lengths of the paths on its own does not affect the utility. Thus, we focus on the algorithm runtime as the path length grows. In Fig. 6, we consider sparse graphs with vertices distributed uniformly with the same number of user data vertices as purpose vertices (dataset 2). We can see that as the path length grows, the runtime in case of BruteForce increases faster.

Lastly, we analyse how the number of vertices in the graph impacts the runtime and the utility of the graph after ap-



(a) 100-vertex graphs (dataset 1a). (b) 1000-vertex graphs (dataset 1b). (c) Dense graphs (dataset 1c).

Figure 4: The number of constraints vs. graph utility after applying the algorithms on graphs from dataset 1.

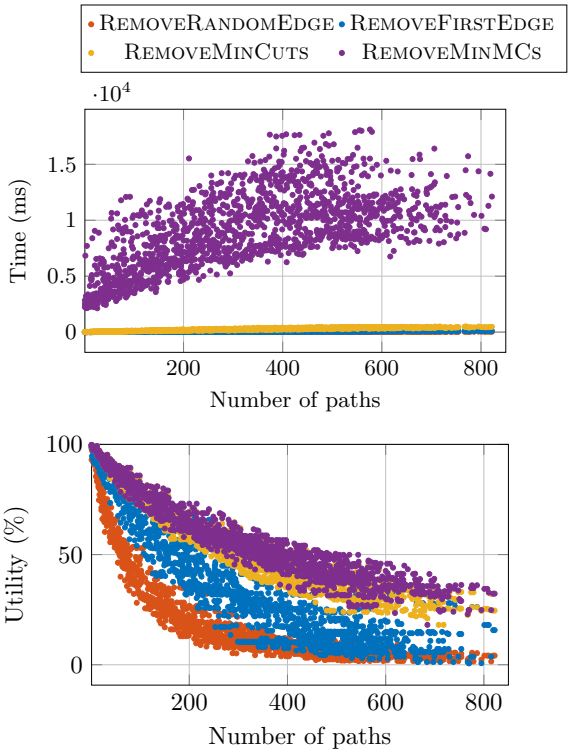


Figure 5: No. of paths vs. runtime and utility (dataset 1c).

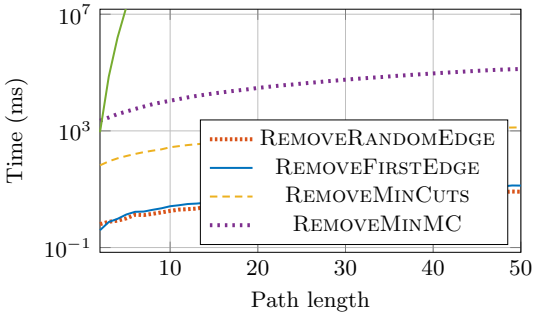


Figure 6: Path length vs. time in sparse graphs (dataset 2).

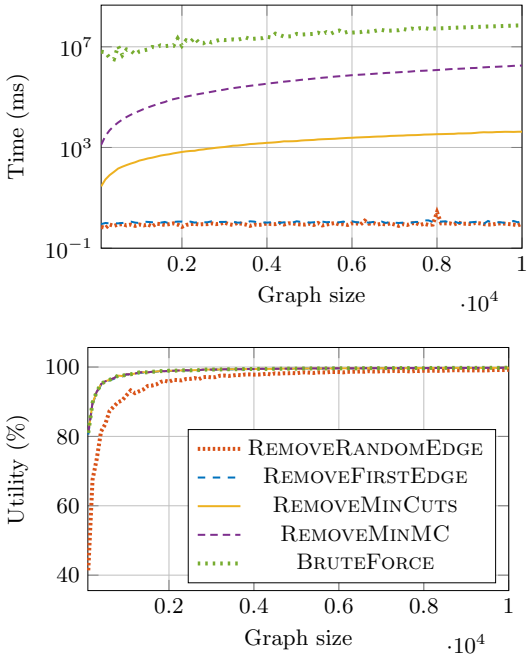


Figure 7: Graph size vs. runtime and utility (dataset 3).

plying the algorithms. To do this, we run the algorithms on graphs from dataset 3. As the number of paths between the constraints and their length are equal for these graphs, in Fig. 7 we can see that the size of the graph has only a slight impact on the execution time for BruteForce. In addition, RemoveMinCuts is faster on average compared to BruteForce and RemoveMinMC. Considering the utility, Fig. 7 shows that the graph size does not have significant impact on utility when the number of paths between the constraints and their length remain equal for the graphs.

8. OPEN PROBLEMS

In this paper, we designed a theoretical mechanism where the data flow is structured as a graph and privacy constraints collected from the user point to pairs of vertices in this graph. While our approach is effective and finds a nearly optimal solution for large graphs, it also creates a range of open problems and challenges.

First, for our algorithms to deliver value, a service provider needs to build an accurate graph. In particular, for large-scale systems with thousands of microservices built on billions lines of code in multiple languages, identifying data flows and all the purposes they are processed for, is challenging and time-consuming. Therefore, future work should develop automated tools to support graph generation.

Second, some of our effective algorithms rely on the simplifying assumption that the value of different information sources is additive. While this is a reasonable starting point and can be a reasonable approximation in some settings, in practice the value may be subadditive (e.g. in the case of redundant data) or superadditive (when data complements each other). As data processing systems keep expanding, future work should focus on exploring more variable models that align the utility and valuation functions with real-world use cases. While the algorithms proposed in this article conceptually generalize to more complex settings, models with different utility and valuation functions may open opportunities for designing more efficient algorithms. For example, in certain machine learning as well as statistical algorithms there is already work that addresses masking or distorting the input with noise. This gives rise to an exciting generalisation of our framework, where utility is affected not by removing the input edge but by distorting the input, and in the future we plan to explore this.

Third, we show that the problem in general is NP-hard. This result provides us with the lower bound on the complexity of the problem. The upper bound, however, depends largely on the complexity of the selected valuation and utility functions. In the future, we plane to investigate the upper bound of the problem in different settings.

In addition, there are several open problems regarding the scalability of the problem. Currently, our solution needs to be recomputed every time a new user enters the system or when an existing user updates their constraints. What is more, every time a change is made, some of the algorithms that process the data would need to be re-run as well, which could be costly. At the same time, there could be many users of the same type, i.e. with similar privacy constraints, and a limited number of different user types which can be known in advance. To take advantage of this, users of the same type could, e.g., be treated as a single user to cope with thousands and even millions of users. This way, if new users enter the system, a new solution can be found quickly. Generally, as more and more users have privacy constraints, new methods are needed that take into account scalability by re-using some of the computation performed for the previous solutions, as well as the costs of making changes.

Finally, there are plenty of opportunities to consider richer types of privacy constraints and user preferences. For example, users may have constraints on combinations of different data types for a specific purpose (e.g. a user may say '*I'm okay with you using my data for advertising, but don't combine my location with my purchase history*') or time restrictions on data processing (*'I'm okay with you sharing my purchase history, but I don't want them to keep it for more than 30 days*'). Future work should formulate new problems around these constraints, as well as constraint the secondary re-use of data later in the data lifecycle.

9. RELATED WORK

With the scale of data collection and processing growing vastly in the recent years, there has been an emergence of initiatives and tools that aim to aid users in controlling the flow of their personal data. Early efforts include the Platform for Privacy Preferences (P3P) which aimed to enable machine-readable privacy policies [7]. Such privacy policies could be automatically retrieved by Web browsers and other tools that can display symbols, prompt users, or take other appropriate actions. Users were able to communicate their privacy constraints to these so-called *user agents* as a list of rules expressed in a P3P Preference Exchange Language (APPEL) [7]. The agents were then able to compare each policy against the user's constraints and assist the user in deciding when to exchange data with websites [7].

However, the P3P lacked a mechanism that would allow for enforcement of the privacy policy within the enterprise and for management of users' individual privacy preferences [6, 3, 17]. Thus, a new approach was proposed [6] where service providers would publish privacy policies, collect and manage user preferences and consent, and enforce the policies throughout their systems. Based on this framework, the Platform for Enterprise Privacy Practices (E-P3P) was developed [5, 16]. Our work builds on this idea and proposes how to satisfy the users' individual preferences *optimally*.

Nonetheless, more general approaches that go beyond the P3P were required to address the need for policy enforcement. To that end, Hippocratic databases were proposed [3, 4, 1, 19, 2] as 'database systems that take responsibility for the privacy of data they manage'. There, personal data was associated with the purposes it was collected for, and metadata such as retention period. Given privacy constraints, the so-called Privacy Constraint Validator would check if the policy is acceptable to the user. Recently, an approach to enable even more fine-grained control of such policies within relational databases was presented [18]. Here, we extend this approach.

Complementing our work, related work has proposed methods for privacy policy-compliant data processing. For example, once it is known how the user's privacy constraints can be optimally satisfied (e.g., using our proposed approach), it is possible to ensure that the datasets used by the data-processing algorithms align with these constraints [8, 9]. Moreover, data processing techniques can be selected based on the consented types of the input data [24]. Furthermore, to make sure that the policies are enforced, a system for checking data usage policies automatically at query runtime has been proposed [23].

10. ACKNOWLEDGEMENTS

E. Gerding was partially funded by the EPSRC-funded platform grant "AutoTrust: Designing a Human-Centred Trusted, Secure, Intelligent and Usable Internet of Vehicles" (EP/R029563/1). G. Konstantinidis was partially funded by the UKRI Horizon Europe guarantee funding scheme for the Horizon Europe projects RAISE (No. 101058479) and UPCAST (No. 101093216).

11. REFERENCES

[1] R. Agrawal, R. Bayardo, C. Faloutsos, J. Kiernan, R. Rantzau, and R. Srikant. Auditing compliance with a hippocratic database. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, pages 516–527, 2004.

[2] R. Agrawal, P. Bird, T. Grandison, J. Kiernan, S. Logan, and W. Rjaibi. Extending relational database systems to automatically enforce privacy policies. In *21st International Conference on Data Engineering (ICDE'05)*, pages 1013–1022. IEEE, 2005.

[3] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. Hippocratic databases. In *Proceedings of the 28th VLDB Conference*, pages 143–154, United States, 2002. VLDB Endownment.

[4] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. Implementing p3p using database technology. In *Proceedings 19th International Conference on Data Engineering (Cat. No. 03CH37405)*, pages 595–606. IEEE, 2003.

[5] P. Ashley, S. Hada, G. Karjoth, and M. Schunter. E-p3p privacy policies and privacy authorization. In *Proceedings of the 2002 ACM Workshop on Privacy in the Electronic Society*, pages 103–109. ACM, 2002.

[6] P. Ashley, C. Powers, and M. Schunter. From privacy promises to privacy management: a new approach for enforcing privacy throughout an enterprise. In *Proceedings of the 2002 Workshop on New Security Paradigms*, pages 43–50. ACM, 2002.

[7] L. F. Cranor. *Web privacy with P3P*. O'Reilly Media, Inc., 2002.

[8] C. Debruyne, H. J. Pandit, D. Lewis, and D. O'Sullivan. Towards generating policy-compliant datasets. In *2019 IEEE 13th International Conference on Semantic Computing (ICSC)*, pages 199–203. IEEE, 2019.

[9] C. Debruyne, H. J. Pandit, D. Lewis, and D. O'Sullivan. "just-in-time" generation of datasets by considering structured representations of given consent for gdpr compliance. *Knowledge and Information Systems*, 62(9):3615–3640, 2020.

[10] Y. Dinitz. Dinitz'algorithm: The original version and Even's version. In O. Goldreich, A. L. Rosenberg, and A. L. Selman, editors, *Theoretical Computer Science: Essays in Memory of Shimon Even*, pages 218–240. Springer, 2006.

[11] J. Edmonds and R. M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM*, 19(2):248–264, 1972.

[12] European Parliament and the Council. Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation). *Official Journal of the European Union*, pages 1–88, May 2016.

[13] D. Filipczuk, E. H. Gerding, and G. Konstantinidis. Consent management in data workflows: A graph problem. In *Proceedings of the 26th International Conference on Extending Database Technology (EDBT)*, 2023.

[14] A. Ghorbani and J. Zou. Data Shapley: Equitable valuation of data for machine learning. In *International Conference on Machine Learning*, pages 2242–2251, Long Beach, California, USA, 2019. PMLR.

[15] D. Huye, Y. Shkuro, and R. R. Sambasivan. Lifting the veil on {Meta's} microservice architecture: Analyses of topology and request workflows. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 419–432, 2023.

[16] G. Karjoth, M. Schunter, and M. Waidner. Platform for enterprise privacy practices: Privacy-enabled management of customer data. In *2nd Workshop on Privacy-Enhancing Technologies. Lecture Notes in Computer Science*, pages 69–84. Springer, 2002.

[17] J. H. Kaufman, S. Edlund, D. A. Ford, and C. Powers. The social contract core. In *Proceedings of the 11th International World Wide Web Conference (WWW)*, pages 210–220, Honolulu, Hawaii, 2002. ACM.

[18] G. Konstantinidis, J. Holt, and A. Chapman. Enabling personal consent in databases. *Proceedings of the VLDB Endowment*, 15(2):375–387, October 2021.

[19] K. LeFevre, R. Agrawal, V. Ercegovac, R. Ramakrishnan, Y. Xu, and D. DeWitt. Limiting disclosure in Hippocratic databases. In *Proceedings of the 30th International Conference on Very Large Databases*, pages 108–119. VLDB Endowment, 2004.

[20] C. Li, D. Y. Li, G. Miklau, and D. Suciu. A theory of pricing private data. *ACM Transactions on Database Systems*, 39(4):1–28, 2014.

[21] R. T. Marler and J. S. Arora. Survey of multi-objective optimization methods for engineering. *Structural and Multidisciplinary Optimization*, 26(6):369–395, 2004.

[22] L. Sion, K. Yskout, D. Van Landuyt, and W. Joosen. Solution-aware data flow diagrams for security threat modeling. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, pages 1425–1432, 2018.

[23] P. Upadhyaya, M. Balazinska, and D. Suciu. Automatic enforcement of data use policies with DataLawyer. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 213–225. ACM, 2015.

[24] Y. Wang and A. Kobsa. Respecting users' individual privacy constraints in web personalization. In *International Conference on User Modeling*, pages 157–166. Springer, 2007.

Technical Perspective: Synthetic Data Needs a Reproducibility Benchmark

Xi He
University of Waterloo
xi.he@uwaterloo.ca

Synthetic data is a vital substitute for real sensitive personal data in supporting social science research and policy studies. Extensive prior research has delved into various models for generating synthetic data, from traditional statistical approaches to cutting-edge deep-learning methods. However, selecting the most suitable one for unforeseen applications poses a significant challenge due to the varying strengths and weaknesses, dependent on factors such as the application domain, data distribution, analytical requirements, and privacy considerations.

Differential privacy (DP) synthesizers have emerged as a prominent approach for generating synthetic data, offering strong mathematical privacy guarantees. These synthesizers first learn a model from real data, inject noise to achieve DP, and then sample the noisy model to generate synthetic datasets. Despite DP offering stronger privacy assurances than its predecessors, such as k-anonymity, DP synthesizers face many utility concerns and criticisms. The concerns are particularly pertinent in real-world applications, such as the U.S. Census's 2020 release, where noise in the data could lead to inaccurate or biased outcomes for critical decisions like allocating block grants.

However, do these concerns apply to non-DP synthesizers, such as the census release before 2020? They likely do, especially if they offer comparable privacy protection, yet there is limited evidence regarding the utility of any synthesizers in practical settings. Common utility proxies for synthetic data evaluation, like descriptive statistics and classification accuracy, offer some procedural representation of analysis tasks but lack validation for real-world applicability. Addressing this gap requires the development of realistic utility benchmarks, a pressing and outstanding problem.

In their article "Epistemic Parity: Reproducibility as an Evaluation Metric for Differential Privacy," the authors propose an evaluation methodology for synthetic data centered on a question: "Can a DP synthesizer produce private tabular data that can be used to derive scientific findings?" This methodology directly addresses practitioners' experience with synthetic data in real-world scenarios, measuring the likelihood that published findings would have changed had the authors used synthetic data, a condition termed epistemic parity. The authors begin by reproducing findings found in peer-reviewed papers using real datasets and then replicate these on their DP synthetic version for comparison.

While this idea may seem straightforward, its execution demands significant effort due to various challenges. Firstly, reproducing conclusions from peer-reviewed papers often encounters low success rates due to factors like unclear computational details and data versioning issues. Second, crafting an effective benchmark to capture real-world analysis complexity and ensure fair comparisons among DP synthesizers requires meticulous navigation and filtering of vast datasets and their studies. Furthermore, each DP synthesizer entails multiple sources of randomness, complicating the task of confidently and fairly reporting evaluation scores. This paper is the first to tackle all these challenges and present a practical taxonomy for reproducing statistical analyses in peer-reviewed publications and a software benchmark package, SynRD, that automates the epistemic parity evaluations for DP synthesizers.

The authors leveraged concepts from reproducibility literature to carefully select datasets from ICPSR, a data repository for social science (consisting of over 100,000 publications spanning 17,312 studies), identify conclusions in the papers, extract relevant findings, and implement corresponding statistical tests. The SynRD benchmark comprises eight datasets, each rigorously reviewed by two researchers with expertise in computing science, statistics, or both, entailing a minimum of thirty hours of work. This effort yielded over a hundred reproducible empirical findings. The selected datasets and findings exhibit diverse properties and characteristics (sample size, number of variables, domain size, outliers, mutual information, skewness, and sparsity) and cover various computational methods such as regression, causal paths, etc. Using this new benchmark on five state-of-the-art DP synthesizers, the authors recover the main results from prior benchmark papers that use simple utility proxies. There are also new and surprising insights, such as the low sensitivity to the privacy budget on reproducibility and the failure of all synthesizers for specific challenging datasets.

This open-sourced and extensible benchmark will continue to grow with new characteristics for evaluating synthetic data, such as its false discovery rates and balance between utility and privacy. SynRD has the possibility to become one milestone benchmark that advances DP synthesizers and other practical synthesizers beyond DP research. If you are intrigued by the construction of SynRD and the new insights into the latest DP synthesizers, or if you aspire to develop cutting-edge DP synthesizers or contribute to synthetic data benchmarks, we highly recommend reading this paper.

Epistemic Parity: Reproducibility as an Evaluation Metric for Differential Privacy

Lucas Rosenblatt*
New York University
New York, NY, USA
lucas.rosenblatt@nyu.edu

Bernease Herman
University of Washington
Seattle, WA, USA
bernease@uw.edu

Anastasia Holovenko
Ukrainian Catholic University
Lviv, Ukraine
anastasia.holovenko@ucu.edu.ua

Wonkwon Lee
New York University
New York, NY, USA
wl2733@nyu.edu

Joshua Loftus
London School of Economics
London, UK
J.R.Loftus@lse.ac.uk

Elizabeth McKinnie
Microsoft
Seattle, WA, USA
Elizabeth.McKinnie@microsoft.com

Taras Rumezhak
Ukrainian Catholic University
Lviv, Ukraine
rumezhak@ucu.edu.ua

Andrii Stadnik
Ukrainian Catholic University
Lviv, Ukraine
andrii.stadnik@ucu.edu.ua

Bill Howe
University of Washington
Seattle, WA, USA
billhowe@uw.edu

Julia Stoyanovich
New York University
New York, NY, USA
stoyanovich@nyu.edu

ABSTRACT

Differential privacy (DP) data synthesizers are increasingly proposed to afford public release of sensitive information, offering theoretical guarantees for privacy (and, in some cases, utility), but limited empirical evidence of utility in practical settings. Utility is typically measured as the error on representative proxy tasks, such as descriptive statistics, multivariate correlations, the accuracy of trained classifiers, or performance over a query workload. The ability for these results to generalize to practitioners' experience has been questioned in a number of settings, including the U.S. Census. In this paper, we propose an evaluation methodology for synthetic data that avoids assumptions about the representativeness of proxy tasks, instead measuring the likelihood that published conclusions would change had the authors used synthetic data, a condition we call epistemic parity. Our methodology consists of reproducing empirical conclusions of peer-reviewed papers on real, publicly available data, then re-running these experiments a second time on DP synthetic data and comparing the results.

We instantiate our methodology over a benchmark of recent peer-reviewed papers in the social sciences. We express the authors' claims computationally to automate the experiment, generate DP synthetic datasets using multiple state-of-the-art mechanisms, then estimate the likelihood that these conclusions will hold. We find that, for reasonable privacy regimes, DP synthesizers can achieve high epistemic parity for several papers in our benchmark. However, some papers, and particularly some specific findings, are difficult to reproduce for any of the synthesizers. Given these results, we recommend a new class of mechanisms that offer stronger utility guarantees (as measured by epistemic parity) and more nuanced privacy protection using application-specific risks and threat models.

*Rosenblatt is the first author, Howe and Stoyanovich are the senior authors.

 This is a minor revision of the paper entitled "Epistemic Parity: Reproducibility as an Evaluation Metric for Differential Privacy," published in PVLDB, Vol. 16, No. 11, ISSN 2150-8097. DOI: https://doi.org/10.14778/3611479.3611517

1. INTRODUCTION

Differential privacy (DP) has been studied intensely for over a decade, and has recently enjoyed uptake in both the private and public sectors. In situations where the downstream analysis is known, one can design specialized mechanisms with high utility [37, 38]. But an active research area is to design general DP data synthesizers (henceforth, synthesizers) that model the entire data distribution, inject noise, then sample the noisy model to generate synthetic datasets intended to be broadly usable in a variety of unanticipated applications. Evidence to support claims of general utility is typically presented as results on proxy tasks over common public datasets (e.g., the ubiquitous Adult dataset [33]). Proxy tasks may include descriptive statistics, queries involving one or two variables [25, 24, 50, 51], classification accuracy [12, 50, 56], and information theoretic measures [56]. Although these proxy tasks are procedurally representative of real tasks, the implicit claim of generalization to practice is rarely explored.

Limited empirical evidence on relevant tasks undermines trust in the practical use of DP. The US Census Bureau adopted DP for disclosure avoidance in the 2020 census, interpreting federal law (the Census Act, 13 U.S.C. § 214, and the Confidential Information Protection and Statistical Efficiency Act of 2002) as a mandate to use advanced methods

to protect against computational reconstruction attacks unforeseen when the laws were passed. But the adoption of DP for the Census was met with resistance among many in the research community, who contend that data infused with DP noise affects demographic totals [47] and exacerbates underrepresentation of minorities [32, 21]. Besides the research implications, there are potential consequences for policy: Block grants are allocated based on minority populations as measured by the census data, and underrepresentation can lead to underfunding integral services including Medicaid, Head Start, SNAP, Section 8 Housing vouchers, Pell Grants, and more [8]. Although the Census Bureau held workshops, released demonstration datasets, and published technical reports to support the community, these outreach efforts realized limited success; multiple lawsuits are still pending as of May 2023.

Despite these challenges, DP still offers stronger guarantees of disclosure protection than, and similar utility to, alternative proposals (e.g., k-anonymity, swapping [8]). DP, when used correctly, ensures that any inferences conducted on data do not reveal whether a single individual's information (including, for example, their gender or race) was included in the data for analysis [15]. DP can therefore not only protect privacy, but also enable access to protected demographic attributes necessary for research on fairness and equity in machine learning [29].

Characterizing DP Error.

A practical method of operationalizing DP is to learn a (noise-infused) model of a dataset, then sample that privatized model to generate synthetic data that can be released publicly [16, 22, 52, 45, 54, 37]. Ideally, this approach would provide a drop-in replacement for the original data that can be used in *any* downstream context to produce reasonably faithful results with strong privacy guarantees. But this ideal is unrealizable, both theoretically and practically. Overly accurate estimates of too many statistics are blatantly non-private, affording full reconstruction of the original dataset [13]. For any DP synthetic dataset, some statistics will tend to be faithful to the original data, while others will incur essentially arbitrary error. If the privacy budget is allocated uniformly across features, descriptive statistics of each individual feature will be faithful, but the conditional probabilities and marginals needed to construct the joint probability distribution, which is needed for general inference, will be unreliably noisy, and vice versa. Utility loss may also be non-uniform across subsets of a dataset, in some cases exacerbating inequity and leading to underrepresentation [2, 32] or to error rate disparities [43]. Designers of DP synthesizers must therefore make some kind of educated guess about which tasks should be preserved and which can be ignored. Further, the error introduced by DP methods can and should be incorporated into statistical models explicitly, just as other sources of error are modeled explicitly. However, current DP synthesizers tend not to provide formal descriptions of the error they introduce; this lack of error guarantees is a major drawback of private data release. Our work does not address this limitation, but does help provide an empirical motivation for doing so.

Methodology.

We propose an evaluation methodology for DP synthesizers based on reproducibility: that *published findings on the original dataset should be replicable on a noise-infused dataset.* We identify conclusions in the text of published papers, extract relevant findings supporting those conclusions, implement the corresponding statistical tests using the authors' data, generate synthetic datasets using state-of-the art DP synthesizers, re-apply the statistical tests over the synthetic data, and then determine if the findings still hold. If all findings hold, we say that the DP synthesizer achieves *epistemic parity* for that paper.

We instantiate our methodology over a benchmark of peer-reviewed sociology papers that are based on public data from the Inter-university Consortium for Political and Social Research (ICPSR) repository. We model quantitative results as an inequality between two numbers, for example, "Those using marijuana first (vs. alcohol or cigarettes first) were more likely to be Black, American Indian/Alaskan Native, multiracial, or Hispanic than White or Asian." [19]

Following Errington *et al.* [18], and as is common in the reproducibility literature, our aim was to identify and reproduce a selection of key findings from each paper. For generality, interpretability and simplicity, we consider whether a conclusion holds over synthetic data to be true if the the two quantities are in the same relative order, and do not attempt to measure the change in effect size or the statistical significance of the difference between the original and synthetic result.

Benchmark and results.

ICPSR is an NSF-funded repository for social science data holding over 100,000 publications associated with 17,312 studies. A study typically involves hundreds of variables and supports dozens of papers. Each paper can be considered to be deriving its own dataset (selected variables and selected rows) from the source data of the study. We apply DP methods to synthesize data for these paper-specific, study-derived datasets. ICPSR studies are publicly available by policy, which enables us to instantiate the epistemic parity methodology and develop a benchmark. Notably, there is increasing demand from the ICPSR leadership and community to support keeping sensitive data private, while generating DP synthetic subsets to support reproducibility. Our methodology can be used to respond to this demand.

Paper selection. The benchmark consists of 4 datasets and 8 recent peer-reviewed papers selected for impact, accessibility of the topic to non-experts, recency, and several other criteria. We extracted findings and attempted to reproduce them, following the "same data, different code, different team" approach to reproducibility, encountering challenges commonly reported in that literature including undocumented data versioning, unspecific or incomplete methodologies, and irreconcilable differences between our reproduction and what the authors report. A complete list of papers that we attempted to reproduce, and the issues we encountered, is available in our public GitHub repository. [1]

DP synthesizer selection. We use six state-of-the-art DP synthesizers, namely, MST [37], PrivBayes [56], PATECTGAN [45], AIM [38], PrivMRF [7], and GEM [35] executing each at their recommended settings.

Summary of results. We find that marginals-based and Bayes-net based state-of-the-art DP synthesizers are able to achieve high epistemic parity for five out of eight papers in

[1] `https://github.com/DataResponsibly/SynRD`

our benchmark, but that some papers, and particularly some specific findings, are difficult to reproduce for any of the synthesizers, suggesting a basis for a new benchmark. The papers on which high epistemic parity is achieved use relatively low-dimensional tabular data. However, as we show empirically, large domain and high-dimensional settings are still a bottleneck for increased adoption of DP synthesizers.

Roadmap and Contributions.

We discuss background and relevant DP synthesis methods in Section 2, and then present our contributions: *(i)* the epistemic parity evaluation methodology, based on reproducing qualitative and quantitative empirical findings in peer-reviewed papers over DP synthetic datasets (Section 3); *(ii)* an instantiation of the methodology for eight peer-reviewed social science publications, creating a reusable benchmark for evaluating synthesizers (Section 4); and *(iii)* an experimental evaluation on our benchmark, using five state-of-the-art DP synthesizers (Section 5). We conclude with a discussion of the results, identifying trade-offs and motivating a new class of privacy techniques that favor strong epistemic parity and de-emphasize privacy risk, in Section 6.

2. BACKGROUND

Differential privacy (DP) ensures that altering or removing one record from a given dataset does not significantly affect the outcome of an analysis or query. Intuitively, DP prevents an observer of a private output from drawing conclusions about which specific individuals' information was included in the input. DP is based on the concept of neighboring datasets, where two datasets are neighboring if they differ in a single record. In the scope of the private synthesizers considered by this paper, datasets X and X' are considered neighboring if the removal of a single element x_i from one yields the other (except in the case of PrivBayes; we account for this in our budget allocation). Informally, DP synthesis mechanisms ensure that a synthetic dataset derived from two neighboring datasets will be similar enough as to hide the presence or anbsence of the removed element.

Different mechanisms use different formulations of DP: AIM and GEM both give *concentrated differential privacy* (ρ-zCDP) guarantees [6], while MST, PATECTGAN and PrivMRF give conventional (ϵ, δ)-DP guarantees, and PrivBayes gives an $(\epsilon, 0)$-DP guarantee. As demonstrated by Bun*et al.* [6], an established hierarchy of these guarantees exists: an $(\epsilon, 0)$-DP mechanism gives $\frac{\epsilon^2}{2}$-zCDP, which gives $(\epsilon\sqrt{2\log(1/\delta)}, \delta)$-DP for every $\delta > 0$. In our experiments, all ϵ parameters are translated using these relationships so as to compare at the same relative privacy settings. As is typical, we set δ to be "cryptographically small:" at most $\frac{1}{n}$ for n records, but typically much smaller [17].

Differentially Private Data Synthesis.

We considered five state-of-the-art private data release methods: MST, AIM, PrivMRF, PATECTGAN, PrivBayes and GEM. We acknowledge that many other methods exist for generating DP data [16, 22, 52, 54]. We chose this set informed by recent work [51, 38] showing that, over randomized query workloads on tabular data, MST, AIM and PrivMRF are the highest-performing marginal-based methods, that PrivBayes is the highest-performing Bayes-net-based method, and that PATECTGAN and GEM are the highest-performing deep learning based methods. AIM, PrivMRF and GEM are more recent than MST; they were not included in the recent dedicated DP synthesizer benchmarking survey [51] and are currently considered to be the state-of-the-art DP synthesizers.

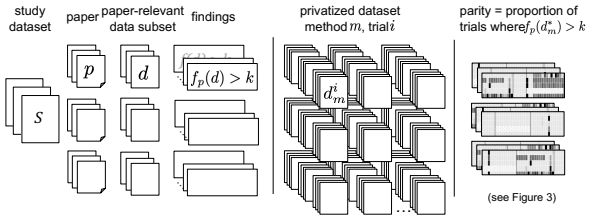


Figure 1: Epistemic parity workflow: Each study dataset supports many papers, each using a subset of the features. The paper's findings are implemented as computable inequalities. We generate many privatized datasets using different random seeds, then compute the proportion of these trials for which the findings hold (Figure 2).

PrivBayes [56] derives a Bayesian model and adds noise to all k-way correlations to ensure differential privacy, and despite being published in 2017 is still competitive with more recent methods. MST [36] relies on the Private-PGM graphical model to construct a maximum spanning tree among attributes in the data feature space, where edges are weighted by mutual information. It can measure 1-, 2- and even 3-way marginals to create a high-fidelity low-dimensional approximation of the joint distribution. AIM [38], like MST, relies on the Private-PGM for parameterizing the underlying distribution, but utilizes an iterative process to take advantage of higher values of ϵ. PrivMRF [7] is another marginal-based algorithm that relies on Private-PGM, and its novelty lies in a clever criteria for the selection of marginals to measure. PATECTGAN [55, 45] relies on a conditional generative adversarial network tuned to tabular data, where the discriminator has privacy constraints. GEM[35] analyzes the "Adaptive Measurements" framework for private synthetic data algorithms, inspired by the MWEM architecture [22], to (1) privately selects a set of queries; (2) obtains noisy measurements of these queries; and (3) updates an approximating distribution according to some loss function.

3. EPISTEMIC PARITY EVALUATION

Intuitively, epistemic parity holds if all published findings from the *original dataset* also hold on the *synthetic dataset*. Consider a finding to be a Boolean condition over the dataset, e.g., whether some statistic f exceeds threshold k. We obtain an epistemic parity score by synthesizing many datasets and reporting the fraction for which the finding holds. Figure 1 illustrates the workflow. The input is a set of papers, and the output is a set of scores indicating whether findings are supported under various DP synthesizers. A study is associated with one dataset and potentially many papers, each using a subset of the variables in the study. We assume public access to the data on which the paper's results were computed; our focus is on evaluating DP methods (requiring ground truth) rather than on protecting the privacy of subjects involved in the study.[2]

[2]Indeed, inaccessible ground truth undermined the US Cen-

Given a paper, we identify natural language claims made by the authors as candidates for findings. Though these claims may appear anywhere in the paper, most were found in the results section. Domain expertise provides an advantage in this task, but we contend that it should always be possible for non-expert readers to identify major claims since the goal of a paper is to communicate findings to a broader audience. For each claim, we identify the quantitative evidence that supports the claim, recording the variables involved and methods used. We then re-implement the analysis to (attempt to) reproduce the salient findings and conclusions in the paper over the original, public dataset.

While this reproduction step is always possible in principle, it can be difficult or impossible in practice [3, 39], and may involve guesswork when the computational details are incomplete. Moreover, inconsistent reproducibility can introduce bias in our benchmark: we may be more likely to include findings for which computational details are clear, which may be those that are simpler to explain or better-known by the author.

If the reproduction was successful, we generate $k \times m$ synthetic datasets representing k trials with different random seeds and m different DP methods, and then draw an additional B samples from each seeded DP method. In our initial benchmark, $k = 10$ and $m = 5$, and $B = 25$. The additional B draws allow us to bootstrap a confidence interval for each (trained) synthesizer. That is, there are two sources of randomness: the training procedure used by the mechanism, and the random sampling of the learned model to actually generate synthetic data. Although each synthetic dataset could be scaled to any number of records — recall that we are sampling a privatized model — we always use the same number of records as the original data for each bootstrap sample. Given this set of synthetic datasets, we again attempt to reproduce the findings using each one. Finally, we contrast the findings based on original and DP data by measuring the proportion of trials, for each method, where a given finding holds. Our methodology is implemented in an open-source framework.

Reproducing Experimental Studies.

We adapt three concepts of reproducibility—*values*, *findings*, and *conclusions*—from Cohen *et al.* [9] into a practical taxonomy for reproducing a statistical analysis in a peer-reviewed publication, and implement a software framework that allows us to conduct concrete experiments around this taxonomy. The atomic element in reproducibility is a *finding*, defined by Cohen *et al.* [9] as "a relationship between the *values* for some reported figure of merit with respect to two or more dependent variables." For the purposes of our study, a *finding* consists of a natural language statement (i.e., a *claim*) reported in a publication, along with evidence provided by one or more quantitative or qualitative sub-statements about the analysis.

Evidence for a *finding* consists of a comparison between two or more *values* that can be evaluated as a Boolean condition. A value may be a scalar (i.e., 34.1%), an aggregated or computed result (i.e., a regression coefficient of 1.2), or even an implicit threshold expressed in natural language (e.g., "a low rate" or "a strong correlation"). In these cases, we instantiate the language as a quantitative threshold, applying conventions from the literature when they exist. For example, a common convention is that Pearson's correlation is considered "strong" when r is larger than 0.7.

A special case of a *finding* is a qualitative *visual finding* that often appears in the form of a figure, table or diagram. A figure encodes many potential *findings*; we do not (necessarily) consider each of these sub-findings on their own in our analysis, but rather treat them as a single *visual finding*: we attempt to reproduce the figure itself, and subjectively evaluate its similarity to the original.

Finally, following Cohen *et al.* [9], a *conclusion* is defined as "a broad induction that is made based on the results of the reported research." A conclusion must be explicitly stated in a paper, and comprises one or several *findings*.

Generating DP Synthetic Data.

Each of the papers that we reproduced using DP synthetic data derived findings from a subset of the full study's data. For example, HSLS:09 consists of over 7000 columns, but Jeong *et al.* [30] used only a subset of 57. We synthesize the subset of data relevant for the reproduced findings and conclusions, as discussed in Section **??**. In the case where a paper relies on longitudinal data from a study, we collapse the data such that it is "one row to one person."

The DP methods for private data release are executed for the range of ϵ values $\epsilon \in \{e^{-3}, e^{-2}, e^{-1}, e^{0}, e^{1}, e^{2}\}$, which represents a small to medium privacy regime [4]. Each DP mechanism is run 10 times to produce, at each ϵ value, $10 \times B$ sampled datasets using the same sample size but different random seeds (where B is the bootstrap parameter). Each DP method involves different hyperparameters and varying levels of tunability, but we use author-recommended settings to avoid biasing results towards our own expertise. We then re-compute the findings for each sample.

If all findings are reproduced regardless of *epsilon* or random seed, we say that the DP mechanism achieves *complete* epistemic parity. But we measure parity as the *proportion* of iterations for which the finding holds. The goal is to overlook small variations in the exact value in favor of maintaining the relative relationships of the computed statistics for interpretability and practical utility.

4. BENCHMARK CONSTRUCTION

In constructing our benchmark, we selected study datasets that have been used in at least 100 papers, focusing on peer-reviewed, publicly available studies from the past 5 years that utilize publicly accessible data and are under 30 pages. We selected: (1) The High School Longitudinal Study (HSLS:09) [10], a longitudinal study of U.S. 9th graders (three of four paper reproduction attempts successful), (2) the National Longitudinal Study of Adolescent and Adult Health (AddHealth) [23] that follows U.S. adolescents from grades 7 through 12 during the 1994-1995 school year (two of four paper reproduction attempts successful), (3) The National Survey on Drug Use and Health (NSDUH) [53] that measures U.S. drug use prevalence and correlates (one of four paper reproduction attempts successful, at least partially due to study variations without clear version records), and (4) the Americans' Changing Lives Survey (ACL) [26], which tracks U.S. adults over time to understand the effects of social connections and work on health (two of two reproduction attempts partially successful), see [44] for details.

sus Bureau's efforts to build trust in DP [5].

Selected Studies.

A study dataset was selected only if it was used in at least 100 papers. For each selected study, we selected peer-reviewed papers published during the past 5 years that are no more than 30 pages long.

HSLS:09: High School Longitudinal Study [10], is a nationally representative, longitudinal study of U.S. 9th graders who were followed through their secondary and postsecondary years.

AddHealth, National Longitudinal Study of Adolescent and Adult Health [23], consists of a nationally representative sample of U.S. adolescents in grades 7 through 12 during the 1994-1995 school year.

NSDUH, National Survey on Drug Use and Health 2004-2014 [53], measures the prevalence and correlates of drug use in the U.S.

ACL, The Americans' Changing Lives Survey [26], is an ongoing longitudinal study of the lives of U.S. adults. The study has several waves, the first of which was conducted in 1986, and each wave continues with the same respondents to determine how social connections, work, and other factors affect health throughout their lifetimes.

Selected Papers.

We will briefly outline each paper from our benchmark. Saw *et al.* [48] utilized HSLS:09 for examining disparities in STEM career aspirations among high school students. Lee *et al.* [34] evaluated the impact of teacher support and self-perceptions on math performance using HSLS:09. Jeong *et al.* [30] investigated racial bias in the performance of machine learning classification tasks with HSLS:09. Fruiht and Chan [20] explored the impact of mentors on first-generation college students using AddHealth. Iverson and Terry [27] analyzed the effects of high school football on later-life depressive and suicidal tendencies using AddHealth. Fairman *et al.* [19] investigated early marijuana use and its consequences using NSDUH. Assari and Bazargan [1] studied the impact of obesity on mortality risk due to cerebrovascular disease using ACL. Pierce and Quiroz [40] examined the effects of social support on emotional states using ACL.

Note on study/dataset dimensionality. We did not explicitly filter papers based on the size of the dataset they used. The studies we considered were very high dimensional (many thousands of variables), but the corresponding papers in our benchmark each follow a standard subsetting procedure, where they select a small collection of variables of interest for analysis. Thus, our benchmark datasets are not as high-dimensional as other benchmarks [38].

Comparison to Other DP Benchmarks.

Selected papers represent 8 new datasets. In this section, we adopt a meta-learning perspective [41, 42] to discuss characteristics that differentiate these datasets from typical ML benchmarks [33, 49] used in prior DP studies [24, 45].

In Table 1, we show several properties and meta-features for eight datasets from our benchmark, as well as for two popular datasets from the UCI Machine Learning repository [14], Adult [33] and Mushroom [49].

Number of outliers is calculated as the number of values that fall outside of the second and third quartiles, summed across all numerical variables. Outliers present a challenge for privatization, as they are easily identifiable.

Mutual information (mean, standard deviation) is calculated for each pair of features. DP synthetic data algorithms like PrivMRF, MST, PrivBayes and AIM are, at their core, interested in *preserving* mutual information between features, but this preservation is challenging given the constrained nature of model fitting (often relying on a small set of 2- or 3-way marginal queries) and the addition of noise for privatization.

Skewness (mean, standard deviation) of a sample is calculated according to the formula for adjusted Fisher-Pearson standardized moment coefficient, which is an unbiased estimate that gives similar results to other popular skewness measures for large samples, but can vary for smaller and moderate-sized samples [31]. The regularity of variables in a dataset (the level of assymmetry in the underlying distributions) affects their ease of replication.

Sparsity (mean, standard deviation) is defined as a normalized ratio of the number of samples over the number of unique values. Sparser data may be harder to capture through noisy marginal measurements.

Table 1 illustrates the benchmark covers a wide range of values of these metrics. Interestingly, one of our most challenging datasets to reproduce, Iverson and Terry [27], had the lowest average mutual information score and one of the highest sparsity scores. Many of the synthesizers we test depend on mutual information to select the marginal measurements for distribution learning. Selecting the most relevant 2-way marginals when mutual information is uniformly *low* and there are many features is clearly a challenge. Moreover, Adult, a common challenge dataset, has uniquely skewed distributions, which aligns with prior work suggesting that this dataset is idiosyncratic therefore less appropriate for evaluation and benchmarking [11].

5. RESULTS

Our benchmark consists of eight papers, each evaluated on six synthetic data algorithms for six values of ϵ, for a total of 36 mechanisms for each paper, each repeated with 10 random seeds. We draw 25 samples of size n, where n is the real data sample size, and bootstrap over this set of samples when calculating average parity over our finding set. Benchmarking extensively with DP synthesizers is computationally expensive [38, 45, 51]. Fitting many synthesizers took 100s of compute hours. Training PrivMRF and PATE-CTGAN was done using NYU's Greene High Performance Computing cluster using A100 and RTX8000 NVIDIA GPUs with 80GB and 48GB of RAM respectively. CPUs from that same cluster were used to train AIM, MST, PrivBayes, and GEM. The benchmark itself (assessing parity per paper) was also run on the cluster.

Epistemic parity: overall performance.

Figure 2 shows parity for all findings across all papers, for each of the five synthesizers, with ϵ regimens of e^{-3}, e^{-2}, e^{-1}, e^{0}, e^{1}, and e^{2}. Darker color indicates lower average parity, while lighter indicates higher average parity. Each paper is a block of rectangles, where the x-axis represents *findings* and the y-axis shows the five synthesizers. The crosshatched cells indicate that a synthesizer was unable to fit to a dataset in under 6 hours.

The final row, labeled "real, bootstrap," in Figure 2 shows the results of our Bayesian bootstrapping control procedure (see full version of the paper for details [44]). We note that over 97% of our findings are reproduced in 100% of our

Table 1: Properties and meta-features of the datasets in our benchmark, and of two datasets that are commonly used for DP benchmarking, Adult and Mushroom. Mutual Information, Skewness and Sparsity are the *average* for each of these metrics across all variables in the dataset. Our results reinforce that synthesizers may struggle with large sample sizes (Fairman *et al.*), large domain sizes (Jeong *et al.*), and low mutual information (Iverson and Terry).

Paper	Sample Size	Variables	Domain Size	Outliers	Mutual Info.	Skewness	Sparsity
Assari and Bazargan [1]	3361	16	9.03e+09	9	0.051 ± 0.153	0.563 ± 1.557	0.253 ± 0.231
Fairman *et al.* [19]	**293581**	6	2.03e+05	0	0.255 ± 0.432	0.185 ± 0.462	0.174 ± 0.165
Fruiht and Chan [20]	4173	11	2.20e+05	6	0.104 ± 0.256	0.607 ± 1.694	0.394 ± 0.183
Iverson and Terry [27]	1762	27	5.71e+15	5	**0.004 ± 0.010**	NaN	0.307 ± 0.180
Jeong *et al.* [30]	15054	57	**7.04e+42**	32	0.020 ± 0.026	0.338 ± 2.850	0.261 ± 0.166
Lee *et al.* [34]	14575	9	5.11e+17	5	2.862 ± 1.242	0.080 ± 0.440	0.111 ± 0.156
Pierce and Quiroz [40]	1585	17	7.19e+11	11	0.030 ± 0.050	0.001 ± 1.050	0.146 ± 0.158
Saw *et al.* [48]	20242	9	4.30e+04	3	0.143 ± 0.145	1.291 ± 1.218	0.354 ± 0.171
Adult [33]	32561	15	9.06e+14	96	0.066 ± 0.053	17.455 ± 22.992	0.125 ± 0.164
Mushroom [49]	8124	23	2.44e+14	74	0.199 ± 0.209	6.211 ± 8.955	0.297 ± 0.219

Bayesian bootstrap iterations. For the remaining inconsistent three findings over the bootstrap, it is unfair to expect the private synthesizers to have higher epistemic parity than the bootstrap control.

The overall performance of the synthesizers was impressive. All synthesizers achieved 100% parity for Lee *et al.* [34], and Fruiht and Chan [20]. Besides PrivMRF (which was computationally infeasible to fit to the data), AIM, MST, PrivBayes, PATECTGAN, and GEM achieved 100% parity for Pierce and Quiroz [40] as well. Both Saw *et al.* [48], and Assari and Bazargan [1] also had very high levels of parity between findings on real and on synthetic data, although each of these papers had at least one finding that was difficult to reproduce.

Two of the papers provided the greatest challenge, and the most interesting results, across privacy regiments and synthesizer types: Fairman *et al.* [19], and Iverson and Terry [27]. These papers were challenging for very different reasons. Fairman *et al.* [19] had the second-smallest domain size, and the fewest variables. However, it had by far the largest sample, consisting of nearly 300K records. This combination made it very sensitive to noise in marginal measurements (as they are essentially counts), in turn making the findings difficult to replicate in low-privacy settings. Still, PrivBayes and MST exhibited impressive performance in comparison to AIM, PATECTGAN and GEM. On the other hand, Iverson and Terry [27] had both one of the largest domains and the most variables of the papers in our benchmark, as well as a low mutual information between variables. No synthesizer with the exception of GEM exhibited convincing parity performance on this paper.

GEM was the strongest performing synthesizer on one paper (Iverson and Terry [27]). For the other papers in our benchmark, neither PATECTGAN nor GEM were the strongest performing. However, these methods were the most computationally tractable on high-dimensional large-domain data, and were the only methods that were feasible to run on Jeong *et al.* [30], where they both achieved 100% parity. Interestingly, PrivBayes often outperformed MST on our benchmark. We believe that this can be explained by two factors: (1) MST is tailored to work on high-dimensional datasets such as NIST, where explicitly parameterizing a conditional structure (like PrivBayes does) is costly and unstable, while the datasets in our benchmark are relatively low-dimensional; and (2) the findings that comprise the epistemic parity metric are based on conclusions that often rely on conditional relationships, which PrivBayes represents explicitly, while MST does not.

PrivMRF was the slowest synthesizer to run, and required a GPU. This requirement limited our ability to fully assess the capabilities of PrivMRF, although we observe that it performed well on the datasets on which it was able to run successfully. PrivBayes was the second-slowest method to run, due to a known limitation in handling high-dimensional data, but performed competitively on datasets on which it was able to run successfully. Notably, no synthesizer succeeded across all papers, and, remarkably, some findings were *never* reproduced by any of the synthesizers.

Epistemic parity across ϵ values.

Figure 3 compares synthesizer performance across reasonable ϵ values, shown on the x-axis in both sub-figures. The left side of the figure shows aggregated epistemic parity as the percentage of reproduced findings on the y-axis, over all iterations of each synthesizer, averaged over all publications in our benchmark. We observe that synthesizer performance (average parity) improves — although not substantially — for higher ϵ values for marginals-based methods PrivMRF, MST, and AIM. At the smallest values ($\epsilon = e^{-3}, e^{-2}$), the performance of PrivBayes, AIM, and MST all begin to noticeably (and understandably!) degrade, especially on certain findings (e.g., 16-21). Interestingly, PrivBayes achieves best performance at $\epsilon = e$, and PATECTGAN and GEM appear insensitive to the value of ϵ. These trends are consistent with the observations in Figure 2, and support the choice of $\epsilon = e$ as a reasonable privacy budget. Overall, we observe that restricting the privacy budget to $\epsilon = e^{-3}$) does not significantly affect the ability of the synthesizers to reproduce the "easy" findings, while increasing it to $\epsilon = e^2$ does not help with reproducing the "difficult" findings. We conjecture that the modeling structure employed by the synthesizer is more important than the scale of private noise.

The right side of Figure 3 shows average variance of epistemic parity. We observe that variance is lowest for PrivMRF, followed by PrivBayes. Further, we observe that the value of ϵ has little impact on parity variance; AIM is the only synthesizer that benefits from a higher value of ϵ in terms of reduced average parity variance.

The observation that epistemic parity is insensitive to ϵ is significant. It suggests that our metric is substantially different compared to other metrics that were previously used for assessment of DP synthesizers. Parity may provide in-

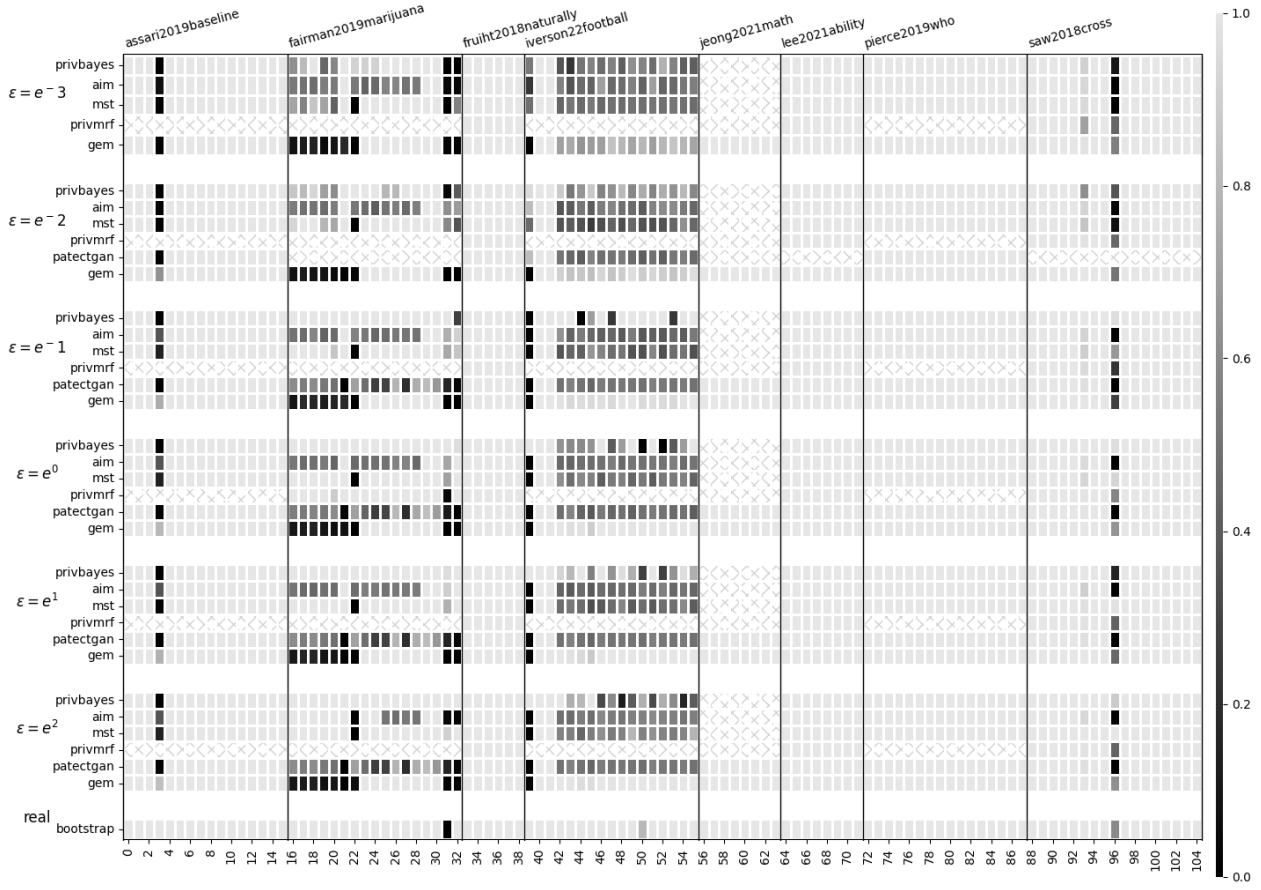


Figure 2: Epistemic parity for six competitive mechanisms for synthesizing data across four ϵ values (e^{-3},e^{-2},e^{-1}, e^{0}, e^{1}, e^{2}). All mechanisms achieve perfect parity on Fruiht and Chan and Lee *et al.*, and all but one achieve perfect parity on Pierce and Quiroz. Only PATECTGAN can scale to support Jeong *et al.*. All methods struggled with the high dimensionality of Iverson and Terry. PrivMRF was too slow to be viable; we report results only for $\epsilon = e^{0}$. Only PrivBayes and MST achieved reasonable parity for Fairman *et al.*. For datasets associated with Assari and Bazargan and Saw *et al.*, only one finding was difficult to reproduce, and all methods struggled. Surprisingly, parity is relatively insensitive to ϵ.

sight into a more fundamental question about whether a DP synthesizer's *methodology* — the types of measurements it takes to constitute a synthetic distribution — is appropriate to preserve the statistical properties of the dataset that are necessary to reproduce *findings*.

Epistemic parity across finding types.

Table 2 summarizes the methods used in the publications in our benchmark, each corresponding to a type of finding. We observe *Mean Difference* (both *Between-class* and *Temporal*) is by far the most common finding type, followed by *Coefficient Difference*. Whether a finding can be reproduced over DP synthetic data depends on several factors, including dataset size (as in Fairman *et al.* [19]) and dimensionality (as in Iverson and Terry [27]). However, finding type likely also plays a role: The majority (19 out of 26) of *Mean Difference / Temporal* findings are in these two papers that were difficult to reproduce. However, we must be cautious to interpret this as a trend: the remaining 7 findings of type *Mean Difference / Temporal (FC)* were in Saw *et al.* [48], and they were reproduced successfully by all synthesizers. In what follows, we qualitatively evaluate the impact of finding type (and, possibly, of other properties of the finding) on its reproducibility over DP synthetic data.

That some findings are easier to reproduce than others is unsurprising. Though each synthesizer relies on a fundamentally different approach to replicating the joint distribution across all of the data, they each struggle with high dimensional data. Further, for general-purpose synthetic data, PrivMRF, AIM, MST and PrivBayes prioritize lower dimensional 2- or 3-way relationships among variables, and thus it is unsurprising that simple mean comparison findings are easily preserved by these methods.

We were surprised by the high number of findings across all papers (even those that we were unable to replicate) relying only on 1- or 2-dimensional comparisons: The low dimensionality suggests that earlier empirical studies (including Tao *et al.* [51] and Hay *et al.* [24]) may be suitable as proxy tasks. Targeted improvements to the synthesizers may allow us to simultaneously support high utility for individual

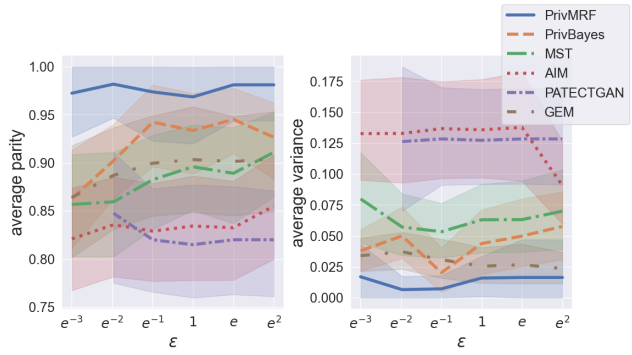


Figure 3: Average epistemic parity across papers achieved by AIM, PrivMRF, MST, PrivBayes, PATECTGAN, and GEM as a function of the privacy parameter $\epsilon \in \{e^{-3}, e^{-2}, e^{-1}, e^{0}, e^{1}, e^{2}\}$. Parity, on the y-axis, is on [0,1] and represents the fraction of reproduced findings over all experiments at each ϵ.

Table 2: Methods used in benchmark papers, each corresponding to a type of *finding* in our framework.

Regression	Descriptive Statistics	8
	Between-Coefficients	4
	Fixed Coefficient (Sign)	2
Causal Paths	Variability	1
	Interaction	1
	Coefficient Difference	19
Logistic Regression	PBR, FNR, FPR	2 (each)
	Accuracy	2
Mean Difference	Between-Class	24
	Temporal (FC)	26
Correlation	Pearson	12
	Spearman	1

findings and their composition into broad conclusions.

Next, we consider 3 findings that were difficult regardless of synthesizer or privacy regimen: #4 (Assari and Bazargan [1]), #39 (Iverson and Terry [27]), and #96 (Saw *et al.* [48]), see Figure 2. Finding #4 is of type *Descriptive Statistics*. It is based on the text statement "Similarly, overall, people had 12.53 years of schooling at baseline (95%CI = 12.34-2.73)." Finding #39 is also of type *Descriptive Statistics*, and is based on a somewhat longer text statement that refers to specific percentages of individuals being diagnosed with specific disorders (5 such pairs of statistics in total). Finding #94 is of type *Mean Difference / Between-class*. It's based on the text statement "From a longitudinal perspective, students from the two lower SES groups—low-middle and low SES groups—had significantly fewer persisters (31.9% and 29.9%) and emergers (6.1% and 5.4%) than their high SES peers (45.1% and 9.0%, respectively)."

These findings were difficult to reproduce because they give specific measurements for variables with large domains. Larger domains require proportionally more DP noise, and so the learned distribution over these variables was too noisy to reproduce the findings within the specified tolerance.

Summary of experimental results.

Overall, we were encouraged by the performance of state-of-the-art synthesizers on our benchmark. DP synthetic data has become more widely used in the social sciences (for Census Data, etc.) and these findings suggest that, in certain contexts, scientists can use DP synthetic data to conduct their scientific inquiry. We caveat this point: *Certain contexts* means relatively low-dimensional tabular data. Our benchmark can be used to assess if those data characteristics hold for a particular dataset, and researchers can proceed with their private analysis with increased confidence.

However, large domain and high-dimensional settings are still a challenge for DP synthesizers: as the domain/number of variables grows, the ease of *fingerprinting* individuals in a dataset increases dramatically. Our findings suggest that existing synthesizers struggle to scale (PrivMRF, MST, AIM, PrivBayes), or are far from achieving reasonable utility (PATECTGAN, GEM). We suggest incorporating more principled methods of data preprocessing, like DP-binning, DP variable pruning, or other domain/variable count reduction techniques into synthesizers, so that successful marginals-based methods can be utilized for more complex data.

6. CONCLUSIONS AND FUTURE WORK

Summary of contributions. We proposed *epistemic parity* as a methodology for measuring the utility of DP synthetic data in support of scientific research. We assembled a benchmark of peer-reviewed papers that analyze one of four studies in the ICPSR social science repository. We then experimentally evaluated epistemic parity achieved by state-of-the-art DP synthesizers over the papers in our benchmark. Overall, we found epistemic parity to be a compelling method for evaluating DP synthesizers. Further, we found that, of the six DP synthesizers we evaluated, no single synthesizer outperformed all others on all papers. Finally, some findings were never reproduced by any of the synthesizers.

Future work: Characterizing false discoveries. Replicating published findings using synthetic versions of the original data can reveal some implications of DP for scientific research. However, this methodology does not assess the possibility of findings that *would have occurred* if the original research had been done on synthetic data, which is related to publication bias [46, 28]. In future work, epistemic parity could be extended to quantify the effect of DP noise in producing these false discoveries by simulating data with both "real" and spurious relationships.

Future work: Rebalancing utility and privacy. Though DP was developed to provide formal guarantees of privacy with best-effort utility, many practitioners and data providers may want the inverse: strong guarantees of utility with quantifiable, flexible risk of privacy violations that can be managed with policy rather than mathematical guarantees. Our benchmark promotes a more holistic discussion of socio-technical-legal systems. Additionally, DP synthesizers can generate arbitrarily large samples at low cost, which makes the power of statistical hypothesis tests another concern for scientific research on private data. Epistemic parity could be extended to estimate the sample size required to achieve a desired power for a particular finding.

7. ACKNOWLEDGEMENTS

This research was supported in part by NSF Awards Nos. 1916505, 1922658, 1934405, and NSF Graduate Research Fellowship Grant No. DGE-2039655, as well as Cisco Award 70618863, the Bill & Melinda Gates Foundation, and the NYU Center for Responsible AI.

8. REFERENCES

[1] S. Assari and M. Bazargan. Baseline obesity increases 25-year risk of mortality due to cerebrovascular disease: role of race. *International Journal of Environmental Research and Public Health*, 16(19):3705, 2019.

[2] E. Bagdasaryan, O. Poursaeed, and V. Shmatikov. Differential privacy has disparate impact on model accuracy. *Advances in neural information processing systems*, 32, 2019.

[3] M. Baker. 1,500 scientists lift the lid on reproducibility. *Nature*, 533(7604), 2016.

[4] C. M. Bowen and F. Liu. Comparative study of differentially private data synthesis methods. *Statistical Science*, 35(2):280–307, 2020.

[5] D. Boyd and J. Sarathy. Differential perspectives: Epistemic disconnects surrounding the us census bureau's use of differential privacy. *Harvard Data Science Review (Forthcoming)*, 2022.

[6] M. Bun and T. Steinke. Concentrated differential privacy: Simplifications, extensions, and lower bounds. In *Theory of Cryptography: 14th International Conference, TCC 2016-B, Beijing, China, October 31-November 3, 2016, Proceedings, Part I*, pages 635–658. Springer, 2016.

[7] K. Cai, X. Lei, J. Wei, and X. Xiao. Data synthesis via differentially private markov random fields. *Proceedings of the VLDB Endowment*, 14(11):2190–2202, 2021.

[8] M. Christ, S. Radway, and S. M. Bellovin. Differential privacy and swapping: Examining de-identification's impact on minority representation and privacy preservation in the us census. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 1564–1564. IEEE Computer Society, 2022.

[9] K. B. Cohen, J. Xia, P. Zweigenbaum, T. J. Callahan, O. Hargraves, F. Goss, N. Ide, A. Névéol, C. Grouin, and L. E. Hunter. Three dimensions of reproducibility in natural language processing. In *LREC... International Conference on Language Resources & Evaluation:[proceedings]. International Conference on Language Resources and Evaluation*, volume 2018, page 156. NIH Public Access, 2018.

[10] B. Dalton, S. J. Ingels, and L. Fritch. High school longitudinal study of 2009 (hsls:09). 2013 update and high school transcript study: A first look at fall 2009 ninth-graders in 2013. nces 2015-037rev. Technical Report ICPSR36423.v1, Inter-University Consortium for Political and Social Research [distributor], 2016. https://doi.org/10.3886/ICPSR36423.v1.

[11] F. Ding, M. Hardt, J. Miller, and L. Schmidt. Retiring adult: New datasets for fair machine learning. *NeurIPS*, 2021.

[12] J. Ding, X. Zhang, X. Li, J. Wang, R. Yu, and M. Pan. Differentially private and fair classification via calibrated functional mechanism. In *AAAI*, volume 34, pages 622–629, 2020.

[13] I. Dinur and K. Nissim. Revealing information while preserving privacy. In F. Neven, C. Beeri, and T. Milo, editors, *Proceedings of the Twenty-Second ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 9-12, 2003, San Diego, CA, USA*, pages 202–210. ACM, 2003.

[14] D. Dua and C. Graff. UCI machine learning repository, 2017.

[15] C. Dwork, F. McSherry, K. Nissim, and A. Smith. Calibrating noise to sensitivity in private data analysis. In *Theory of cryptography conference*, pages 265–284. Springer, 2006.

[16] C. Dwork, M. Naor, O. Reingold, G. N. Rothblum, and S. Vadhan. On the complexity of differentially private data release: efficient algorithms and hardness results. In *Proceedings of the forty-first annual ACM symposium on Theory of computing*, pages 381–390, 2009.

[17] C. Dwork, A. Roth, et al. The algorithmic foundations of differential privacy. *Foundations and Trends in Theoretical Computer Science*, 9(3-4):211–407, 2014.

[18] T. M. Errington, A. Denis, N. Perfito, E. Iorns, and B. A. Nosek. Reproducibility in cancer biology: challenges for assessing replicability in preclinical cancer biology. *Elife*, 10:e67995, 2021.

[19] B. J. Fairman, C. D. Furr-Holden, and R. M. Johnson. When marijuana is used before cigarettes or alcohol: Demographic predictors and associations with heavy use, cannabis use disorder, and other drug-related outcomes. *Prevention Science*, 20(2):225–233, 2019.

[20] V. Fruiht and T. Chan. Naturally Occurring Mentorship in a National Sample of First-Generation College Goers: A Promising Portal for Academic and Developmental Success. 61(3-4):386–397, 2018.

[21] G. Ganev, B. Oprisanu, and E. De Cristofaro. Robin hood and matthew effects–differential privacy has disparate impact on synthetic data. *arXiv preprint arXiv:2109.11429*, 2021.

[22] M. Hardt, K. Ligett, and F. McSherry. A simple and practical algorithm for differentially private data release. *arXiv preprint arXiv:1012.4763*, 2010.

[23] Harris, Kathleen Mullan and Udry, J. Richard. National longitudinal study of adolescent to adult health (add health), 1994-2018 [public use]. Technical Report ICPSR21600.v25, Inter-university Consortium for Political and Social Research [distributor], Carolina Population Center, University of North Carolina-Chapel Hill [distributor], 2022. https://doi.org/10.3886/ICPSR21600.v25.

[24] M. Hay, A. Machanavajjhala, G. Miklau, Y. Chen, and D. Zhang. Principled evaluation of differentially private algorithms using dpbench. In *Proceedings of the 2016 International Conference on Management of Data*, pages 139–154, 2016.

[25] R. Hill. Evaluating the utility of differential privacy: A use case study of a behavioral science dataset. In *Medical Data Privacy Handbook*, pages 59–82. Springer, 2015.

[26] J. S. House. Americans' changing lives: Waves i, ii, iii, iv, and v, 1986, 1989, 1994, 2002, and 2011. Technical Report ICPSR04690.v9, Inter-university Consortium for Political and Social Research [distributor], 2018. https://doi.org/10.3886/ICPSR04690.v9.

[27] G. L. Iverson and D. P. Terry. High school football and risk for depression and suicidality in adulthood: findings from a national longitudinal study. *Frontiers in neurology*, 12, 2021.

[28] S. Iyengar and J. B. Greenhouse. Selection models and the file drawer problem. *Statistical Science*, pages 109–117, 1988.

[29] M. Jagielski, M. Kearns, J. Mao, A. Oprea, A. Roth, S. Sharifi-Malvajerdi, and J. Ullman. Differentially private fair learning. In *ICML*, pages 3000–3008, 2019.

[30] H. Jeong, M. D. Wu, N. Dasgupta, M. Médard, and F. P. Calmon. Whogets the benefit of the doubt? racial bias in machine learning algorithms applied to secondary school math education. In *35th Conference on Neural Information Processing Systems (NeurIPS 2021) Workshop on Math AI for Education (MATHAI4ED)*, 2021.

[31] D. N. Joanes and C. A. Gill. Comparing measures of sample skewness and kurtosis. *The Statistician*, 47:183–189, 1998.

[32] C. T. Kenny, S. Kuriwaki, C. McCartan, E. T. Rosenman, T. Simko, and K. Imai. The use of differential privacy for census data and its impact on redistricting: The case of the 2020 us census. *Science advances*, 7(41):eabk3283, 2021.

[33] R. Kohavi and B. Becker. UCI adult data set. Technical report, UCI Machine Learning Repository, 1996. `https://archive.ics.uci.edu/ml/datasets/adult`.

[34] G. Lee and S. D. Simpkins. Ability self-concepts and parental support may protect adolescents when they experience low support from their math teachers. *Journal of Adolescence*, 88:48–57, 2021.

[35] T. Liu, G. Vietri, and S. Z. Wu. Iterative methods for private synthetic data: Unifying framework and new methods. *Advances in Neural Information Processing Systems*, 34:690–702, 2021.

[36] R. McKenna, G. Miklau, M. Hay, and A. Machanavajjhala. Optimizing error of high-dimensional statistical queries under differential privacy. *arXiv preprint arXiv:1808.03537*, 2018.

[37] R. McKenna, G. Miklau, and D. Sheldon. Winning the NIST contest: A scalable and general approach to differentially private synthetic data. *arXiv preprint arXiv:2108.04978*, 2021.

[38] R. McKenna, B. Mullins, D. Sheldon, and G. Miklau. Aim: An adaptive and iterative mechanism for differentially private synthetic data. *arXiv preprint arXiv:2201.12677*, 2022.

[39] National Academies of Sciences, Engineering, and Medicine and others. Reproducibility and replicability in science. 2019.

[40] K. D. R. Pierce and C. S. Quiroz. Who matters most? social support, social strain, and emotions. *Journal of Social and Personal Relationships*, 36(10):3273–3292, 2019.

[41] F. Pinto, C. Soares, and J. Mendes-Moreira. Towards automatic generation of metafeatures. In J. Bailey, L. Khan, T. Washio, G. Dobbie, J. Z. Huang, and R. Wang, editors, *Advances in Knowledge Discovery and Data Mining*, pages 215–226, Cham, 2016. Springer International Publishing.

[42] A. Rivolli, L. P. F. Garcia, C. Soares, J. Vanschoren, and A. C. P. de Leon Ferreira de Carvalho. Characterizing classification datasets: a study of meta-features for meta-learning. *arXiv: Learning*, 2018.

[43] L. Rosenblatt, J. Allen, and J. Stoyanovich. Spending privacy budget fairly and wisely. *CoRR*, abs/2204.12903, 2022.

[44] L. Rosenblatt, B. Herman, A. Holovenko, W. Lee, J. R. Loftus, E. Mckinnie, T. Rumezhak, A. Stadnik, B. Howe, and J. Stoyanovich. Epistemic parity: Reproducibility as an evaluation metric for differential privacy. *Proc. VLDB Endow.*, 16(11):3178–3191, 2023.

[45] L. Rosenblatt, X. Liu, S. Pouyanfar, E. de Leon, A. Desai, and J. Allen. Differentially private synthetic data: Applied evaluations and enhancements. *arXiv preprint arXiv:2011.05537*, 2020.

[46] R. Rosenthal. The file drawer problem and tolerance for null results. *Psychological bulletin*, 86(3):638, 1979.

[47] S. Ruggles, C. Fitch, D. Magnuson, and J. Schroeder. Differential privacy and census data: Implications for social and economic research. In *AEA papers and proceedings*, volume 109, pages 403–08, 2019.

[48] G. Saw, C.-N. Chang, and H.-Y. Chan. Cross-sectional and longitudinal disparities in stem career aspirations at the intersection of gender, race/ethnicity, and socioeconomic status. *Educational Researcher*, 47(8):525–531, 2018.

[49] J. Schlimmer. UCI adult data set. Technical report, UCI Machine Learning Repository, 1987. `https://archive.ics.uci.edu/ml/datasets/mushroom`.

[50] S. Takagi, T. Takahashi, Y. Cao, and M. Yoshikawa. P3gm: Private high-dimensional data release via privacy preserving phased generative model. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pages 169–180. IEEE, 2021.

[51] Y. Tao, R. McKenna, M. Hay, A. Machanavajjhala, and G. Miklau. Benchmarking differentially private synthetic data generation algorithms. *arXiv preprint arXiv:2112.09238*, 2021.

[52] R. Torkzadehmahani, P. Kairouz, and B. Paten. DP-CGAN: differentially private synthetic data and label generation. *CoRR*, abs/2001.09700, 2020.

[53] United States Department of Health and Human Services. National survey on drug use and health (nsduh), 2014. Technical Report ICPSR36361.v1, Inter-university Consortium for Political and Social Research [distributor], 2016. https://doi.org/10.3886/ICPSR36361.v1.

[54] G. Vietri, G. Tian, M. Bun, T. Steinke, and S. Wu. New oracle-efficient algorithms for private synthetic data release. In *ICML*, pages 9765–9774, 2020.

[55] L. Xu, M. Skoularidou, A. Cuesta-Infante, and K. Veeramachaneni. Modeling tabular data using conditional gan. *Advances in Neural Information Processing Systems*, 32, 2019.

[56] J. Zhang, G. Cormode, C. M. Procopiuc, D. Srivastava, and X. Xiao. Privbayes: Private data release via bayesian networks. SIGMOD 2014.

Learning to Restructure Tables Automatically

Joseph M. Hellerstein
UC Berkeley
hellerstein@berkeley.edu

By now, it is widely-accepted folk wisdom that "half of the time in any data analysis project is spent wrangling the data". Analytic algorithms and tools—built on mathematical foundations of matrices and relations—require their data to be lined up in particular rows and columns. In the relational model (known in data science circles as "tidy data"), each row is an independent observation, and each column is a distinct attribute of the phenomenon described by the data. While there are many thorny aspects to data wrangling, perhaps none is more basic than the challenge of getting data reorganized, positionally, into the right form for analysis.

Unfortunately, most data sets do not arrive tidy, in the sense of being relational. In particular, tables made for human consumption—e.g., spreadsheets or web tables—are notorious for requiring reorganization before analysis. Some tables contain a patchwork of rectangular regions, using colors, separator lines, or metadata embedded inside the data to demarcate ranges of cells. Other tables encode data that we want to analyze into the column names (metadata), making those values difficult to query. Some tables simply lay things out awkwardly: data that we want to analyze together (say, a year of daily temperature readings) is spread across multiple columns (e.g., a column per day), making calculations across the full dataset tedious to express.

Every data set is a little different, so reorganizing data into the right rows and columns typically involves generating bespoke, multi-step wrangling *scripts*. Automating this task is fundamentally a problem of program synthesis, usually in a data-centric domain-specific language (DSL).

The *AutoTables* paper by Li, *et al.* in VLDB '23 is an automation breakthrough in this space, setting a new bar for AI assistance in wrangling messy data tables into relational form. The paper is impressive not only for its results, but for its pragmatism and insight.

As the authors point out in framing the problem, a key challenge is the human bottleneck in learning and inference in this domain. Supervised learning is problematic here: human users are not good at reading tables and classifying their problems. At inference time, standard "query by example" UIs are not tenable: it is extremely tedious to write down a "correct" data table in non-trivial scenarios. Instead, the authors attempt to train models and perform inference *without any human input*. How does that work? That is the big "aha moment" at the heart of this very readable paper.

The key idea comes from the algebra: operations in a data wrangling DSL typically have inverses. Hence every operation that makes some data set cleaner also has an inverse that makes the resulting data set dirtier! In particular, if we start with a clean relational table R (say from a relational database), and "dirty it up" with operation O^{-1}, then we have a labeled example: the resulting table $R' = O^{-1}(R)$ is an example of table that can be cleaned via the operation O. Given that simple observation, the rest follows: a training set can be generated on the fly by sampling from the space of (invertible) transformation operators and their parameters, and a space of clean relational tables to be dirtied. Once a model is trained on that data, there is only one query at inference time: "Relationalize This!"

Much of the remaining material in the paper involves featurizing tables so that well-studied learning techniques can capture the patterns associated with R' vs. R in a loss function. The featurization techniques in the paper relate to data visualization, in the sense that they treat data tables as a special case of images. Intuitively, a clean data table is a grid of cells, which—like pixels in an image—have some coherence. The values down a column of a clean table should certainly share certain features. The values of adjacent cells row-wise will exhibit patterns also. The space of expected patterns may be hard for humans to codify, much like the patterns associated with images of cats. But as we know, deep nets are good at learning and discriminating these patterns. The authors cannily borrow tricks used in training computer vision models—e.g., data augmentation—and show that they are effective here as well.

The paper primarily presents its results in the context of coding environments like R and Python. It strikes me that data wrangling automation of this sort may have even more impact in low-code visual tools—perhaps directly in the venerable spreadsheet, the progenitor of low-code innovation! It will be interesting to watch how improvements in automation like this paper are integrated into the user experience of data analysts across industries and application domains. Regardless of how that plays out, this paper is a big step forward on one of the core problems in data wrangling.

The original version of this paper is entitled "Auto-Tables: Synthesizing Multi-Step Transformations to Relationalize Tables without Using Examples" and was published in *PVLDB 16(11)*, July 2023, VLDB Endowment.)

Auto-Tables: Relationalize Tables without Using Examples

Peng Li*
Georgia Tech
Atlanta, GA
pengli@gatech.edu

Yeye He
Microsoft Research
Redmond, WA
yeyehe@microsoft.com

Cong Yan
Microsoft Research
Redmond, WA
coyan@microsoft.com

Yue Wang
Microsoft Research
Redmond, WA
wanyue@microsoft.com

Surajit Chaudhuri
Microsoft Research
Redmond, WA
surajitc@microsoft.com

ABSTRACT

Relational tables, where each row corresponds to an entity and each column corresponds to an attribute, have been the standard for tables in relational databases. However, such a standard cannot be taken for granted when dealing with tables "in the wild". Our survey of real spreadsheet-tables and web-tables shows that over 30% of such tables do not conform to the relational standard, for which complex table-restructuring transformations are needed before these tables can be queried easily using SQL-based tools. Unfortunately, the required transformations are non-trivial to program, which has become a substantial pain point for technical and non-technical users alike, as evidenced by large numbers of forum questions in places like StackOverflow and Excel/Tableau forums.

We develop an Auto-Tables system that can automatically synthesize pipelines with multi-step transformations (in Python or other languages), to transform non-relational tables into standard relational forms for downstream analytics, obviating the need for users to manually program transformations. We compile an extensive benchmark for this new task, with 244 real test cases collected from user spreadsheets and online forums. Our evaluation suggests that Auto-Tables can successfully synthesize transformations for over 70% of test cases at interactive speeds, without requiring any input from users, making this an effective tool for users to prepare data for downstream analytics.

1. INTRODUCTION

Modern data analytics like SQL and BI are predicated on a standard format of relational tables, where each row corresponds to a distinct "entity", and each column corresponds to an "attribute" for the entities that contains homogeneous data-values. While such tables are de-facto standards in relational databases, to the extent that we as database people may take them for granted, we would like the highlight that a significant fraction of tables "in the wild" actually fail to conform to such standards, making it considerably more challenging to query them using SQL-based tools.

Non-relational tables are common, but hard to query. Real tables in the wild, such as spreadsheet-tables or web-tables, can often be "non-relational" and hard to query, unlike tables that we expect to find in relational databases. We randomly sampled hundreds of user spreadsheets (in Excel), and web tables (from Wikipedia), and found around 30-50% tables to have such issues. Figure 1 and Figure 2 show real samples taken from spreadsheets and the web, respectively, to demonstrate these common issues. (We emphasize that the problem is prevalent at a very large scale, since there are millions of tables like these in spreadsheets and on the web.)

Take Figure 1(a) for example. The table on the left is not a standard relational table, because each column marked in green contains sales numbers for only a single day ("`19-Oct`", "`20-Oct`", etc.), making these column values highly homogeneous in the horizontal direction (while in typical relational tables, we expect values in columns to be homogeneous in the vertical direction). Although this specific table format makes it easy for humans to eyeball changes day-over-day by reading horizontally, it is unfortunately hard to analyze using SQL. Imagine that one needs to compute the 14-day average of sales, starting from "`20-Oct`" – for this table, one has to write: `SELECT SUM("20-Oct", "21-Oct", "22-Oct", ...) FROM T`, across 14 different columns, which is long and unwieldy to write. Now imagine we need 14-day moving averages with every day in October as the starting date – the resulting SQL is highly repetitive and hard to manage.

In contrast, consider a transformed version of this table, shown on the right of Figure 1(a). Here the homogeneous columns in the original table (marked in green) are transformed into only two new columns: "`Date`" and "`Units Sold`", using a transformation operator called "`stack`" (listed in the first row of Table 1). This transformed table contains the same information as the original table, but is much easier to query – e.g., the same 14-day moving average can be computed using a succinct range-predicate on the '`Date`" column, where the starting date "`20-Oct`" is a literal parameter that can be easily changed into other values.

There are many such spreadsheet tables that require different kinds of transformations before they are ready for SQL-based analysis. Figure 1(b) shows another example, where every 3 columns form a repeating group, representing "`Revenue/Units Sold/Margin`" for a different year (marked in red/green/blue in the figure). Tables with these repeat-

*Part of work done while at Microsoft.

 This is a minor revision of the paper entitled "Auto-Tables: Synthesizing Multi-Step Transformations to Relationalize Tables without Using Examples", published in PVLDB, Vol. 16, No. 11, 2150-8097. DOI: https://doi.org/10.14778/3611479.3611534

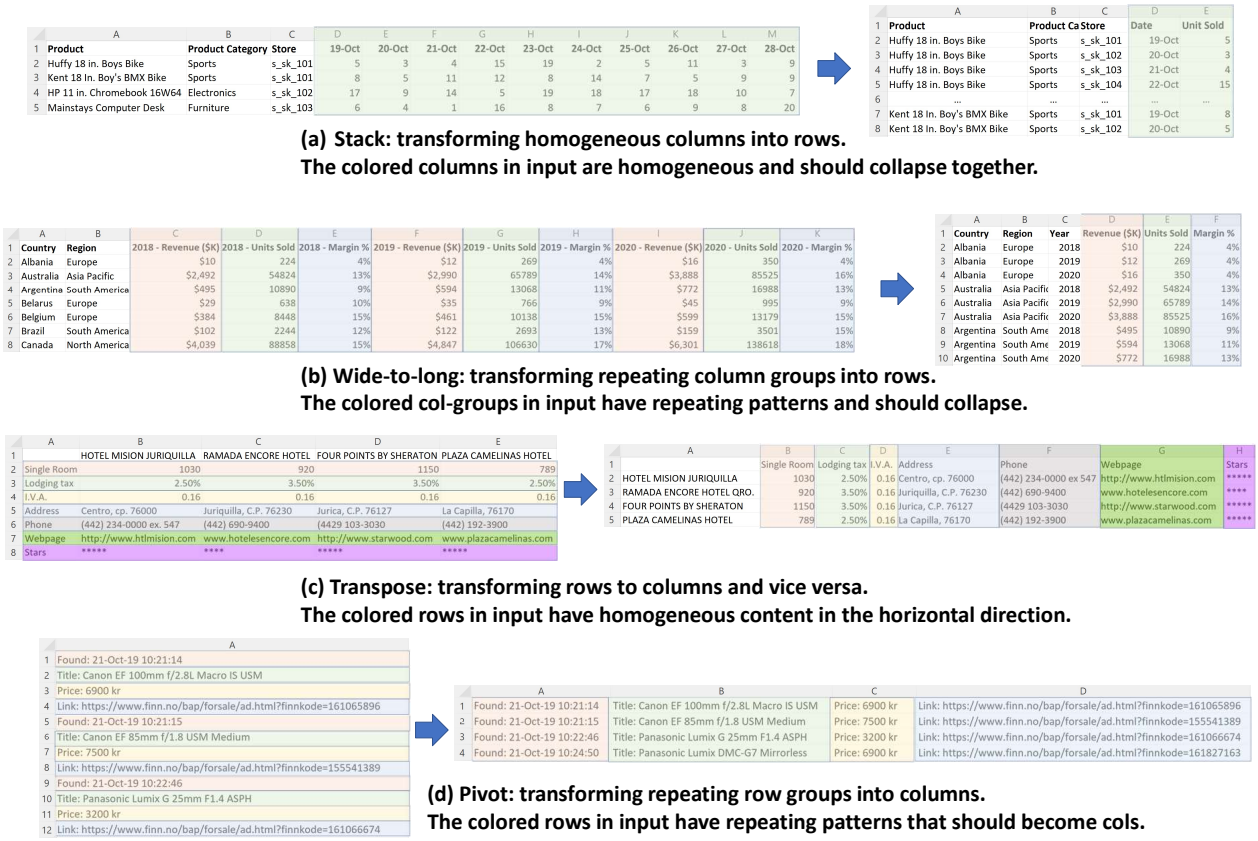


Figure 1: Example input/output tables for 4 operators in AUTO-TABLES: (a) Stack, (b) Wide-to-long, (c) Transpose, (d) Pivot. The input-tables (on the left) are not relational and hard to query, which need to be transformed to produce corresponding output-tables (on the right) that are relational and easy to query. Observe that the color-coded, repeating row/column-groups are "visual" in nature, motivating a CNN-like architecture like used in computer vision for object-detection.

ing column-groups are also hard to query, just like Figure 1(a), but in this case the required transformation operator is called "`wide-to-long`" (second row of Table 1).

Figure 1(c) shows yet another example, where each hotel corresponds to a column (whose names are in row-1), and each "attribute" of these hotels corresponds to a row. Note that in this case values in the same rows are homogeneous (marked in different colors), unlike relational tables where values in the same columns are homogeneous. A transformation called "`transpose`" is required in this case (listed in the third row of Table 1), to make the resulting table, shown on the right of the figure, easy to query – for instance, a query to sum up the total number of hotel rooms is hard to write on the original table, but can be easily achieved using a simple SUM query on the "`Single Room`" column in the transformed table.

Figure 1(d) shows another example where columns are represented as rows in the table on the left. This is similar to Figure 1(c), except that the rows in this case are "repeating" in groups, thus requiring a different transformation operator called "`pivot`" (listed in the fourth row of Table 1) as opposed to "`transpose`". The resulting table is shown on the right, which becomes easy to query.

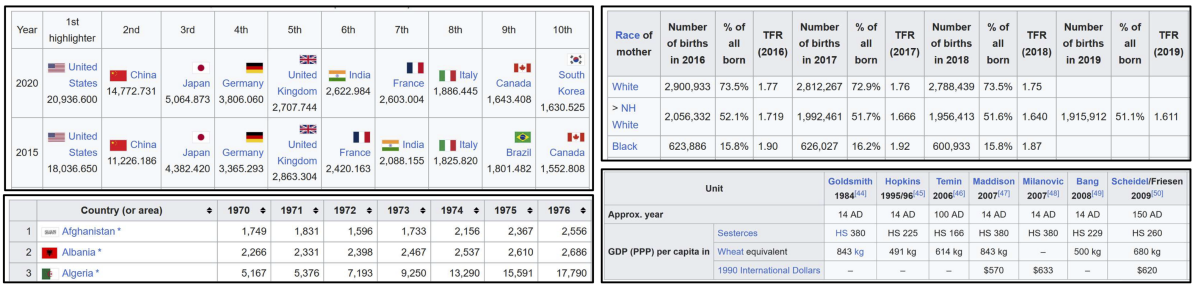
While the examples so far are all taken from spreadsheets, we note that similar structural issues are also widespread in Web tables. Figure 2 shows real examples from Wikipedia, which share similar characteristics as the spreadsheet tables in Figure 1, which all require transformations before these tables can be queried effectively.

Non-relational tables are hard to "relationalize". We mentioned that the example tables in Figure 1 and Figure 2 require different transformation operators. Table 1 shows 8 such transformation operators commonly needed to relationalize tables (where the first 4 operators correspond to the examples we see in Figure 1).

The first column of Table 1 shows the name of the "logical operator", which may be instantiated differently in different languages (e.g., in Python or R), with different names and syntax. The second column of the table shows the equivalent Pandas operator in Python [11], which is a popular API for manipulating tables among developers and data scientists, that readers may be familiar with.

While the operations listed in Table 1 already exist in languages such as R and Python, they are not easy for users to invoke correctly, because users need to:

1. Visually identify different structural issues in an input table that make it hard to query (e.g., repeating row-/column-groups shown in Fig. 1(a-d)), which is not obvious to non-expert users;
2. Map the visual pattern identified from the input table, to a corresponding operator in Table 1 that can handle such issues. This is hard for users not familiar with the exact terminologies to describe these transformations (e.g., `pivot` vs. `stack`);



Year	1st highlighter	2nd	3rd	4th	5th	6th	7th	8th	9th	10th
2020	United States 20,936.600	China 14,772.731	Japan 5,064.873	Germany 3,806.060	United Kingdom 2,707.744	India 2,622.984	France 2,603.004	Italy 1,886.445	Canada 1,643.408	South Korea 1,630.525
2015	United States 18,036.650	China 11,226.186	Japan 4,382.420	Germany 3,365.293	United Kingdom 2,863.304	France 2,420.163	India 2,088.155	Italy 1,825.820	Brazil 1,801.482	Canada 1,552.808

Race of mother	Number of births in 2016	% of all born	TFR (2016)	Number of births in 2017	% of all born	TFR (2017)	Number of births in 2018	% of all born	TFR (2018)	Number of births in 2019	% of all born	TFR (2019)
White	2,900,933	73.5%	1.77	2,812,267	72.9%	1.76	2,788,439	73.5%	1.75			
> NH White	2,056,332	52.1%	1.719	1,992,461	51.7%	1.666	1,956,413	51.6%	1.640	1,915,912	51.1%	1.611
Black	623,886	15.8%	1.90	626,027	16.2%	1.92	600,933	15.8%	1.87			

	Country (or area)	1970	1971	1972	1973	1974	1975	1976
1	Afghanistan *	1,749	1,831	1,596	1,733	2,156	2,367	2,556
2	Albania *	2,266	2,331	2,398	2,467	2,537	2,610	2,686
3	Algeria *	5,167	5,376	7,193	9,250	13,290	15,591	17,790

Unit		Goldsmith 1984[44]	Hopkins 1995/96[45]	Temin 2006[46]	Maddison 2007[47]	Milanovic 2007[48]	Bang 2008[49]	Scheidel/Friesen 2009[50]
Approx. year		14 AD	14 AD	100 AD	14 AD	14 AD	14 AD	150 AD
GDP (PPP) per capita in	Sesterces	HS 380	HS 225	HS 166	HS 380	HS 380	HS 229	HS 260
	Wheat equivalent	843 kg	491 kg	614 kg	843 kg	–	500 kg	680 kg
	1990 International Dollars	–	–	–	$570	$633	–	$620

Figure 2: Real Web tables from Wikipedia that are also non-relational, similar to the spreadsheet tables shown in Figure 1.

Table 1: Auto-Tables DSL: table-restructuring operators and their parameters to "relationalize" tables. These operators are common and exist in many different languages, like Python Pandas and R, sometimes under different names.

DSL operator	Python Pandas equivalent	Operator parameters	Description (example in parenthesis)
stack	melt [14]	start_idx, end_idx	collapse homogeneous cols into rows (Fig. 1a)
wide-to-long	wide_to_long [18]	start_idx, end_idx, delim	collapse repeating col-groups into rows (Fig. 1b)
transpose	transpose [17]	-	convert rows to columns and vice versa (Fig. 1c)
pivot	pivot [15]	repeat_frequency	pivot repeating row-groups into cols (Fig. 1d)
explode	explode [12]	column_idx, delim	convert composite cells into atomic values
ffill	ffill [13]	start_idx, end_idx	fill structurally empty cells in tables
subtitles	copy, ffill, del	column_idx, row_filter	convert table subtitles into a column
none	-	-	no-op, the input table is already relational

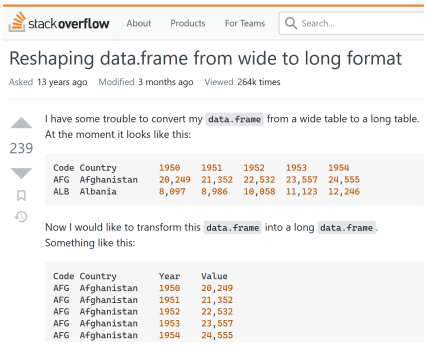


Figure 3: Example user question from StackOverflow, on how to restructure tables. Questions like this are common not only among technical users, but also non-technical users, as similar questions are commonly found on forums for Excel, Power-BI, and Tableau users too [6, 7, 8, 9].

3. Parameterize the chosen operator appropriately (e.g., which columns to collapse, what is the repeating frequency, etc.). This is again hard, as even developers need to consult API documentations that can be long and complex.
4. Certain input tables require more than one transformation step, for which users need to repeat steps (1)-(3).

Completing these steps is a tall order even for technical users, as evidenced by a large number of such questions on forums like StackOverflow. Figure 3 shows one example (popular with many up-votes) – the developer provides example input/output tables to demonstrate the desired transformation, in order to ask what operators should be used.

If technical users like developers find it hard to restructure their tables as these StackOverflow questions would show, it comes as no surprise that non-technical enterprise users, who often deal with tables in spreadsheets, would find the task even more challenging. We find a large number of similar questions on Excel and Tableau forums (e.g., [6, 7, 8, 9]), where users complain that without the required transformations it is hard to analyze data using SQL-based or Excel-based tools (e.g., [2, 3, 4, 5]). The prevalence of these questions confirms table-restructuring as a common pain point for both technical and non-technical users.

Auto-Tables: synthesize transformations without examples. In this work, we propose a new paradigm to automatically synthesize table-restructuring steps to relationalize tables, using the operators in Table 1, *without requiring users to provide examples*. Our key intuition of why we can do away with examples in our task, lies in the observation that given an input table, the logical steps required to relationalize it are *almost always unique*, as the examples in Figure 1 would all show. This is because the transformations required in our task only "restructure" tables, that do not actually "alter" the table content, which is unlike prior work that focuses on *row-to-row transformations* (e.g., TDE [31] and FlashFill [29]), or SQL-by-example (e.g. [22, 50]), where the output is "altered" that can produce many possible outcomes, which would require users to provide input/output examples to demonstrate the desired outcome.

For our task, we believe it is actually important *not* to ask users to provide examples, because in the context of table-to-table transformations like in our case, asking users to provide examples would mean users have to specify *an output table*, which is a substantial amount of typing effort, making it cumbersome to use.

As humans, we can "visually" recognize rows/columns patterns (e.g., homogeneous value groups, as color-coded vertically and horizontally in Figure 1), to correctly predict which operator to use. The question we ask in this paper, is whether an algorithm can "learn" to recognize such patterns by scanning the input tables alone, to predict suitable transformations, in a manner that is analogous to how computer-vision algorithms would scan a picture to identify common but more complex objects like dogs and cats.

In computer vision, in order to pick up subtle clues from

pictures, object detection algorithms are typically trained using large amounts of labeled data [26] (e.g., pictures of dogs that are manually labeled as such). In our task, we do not have such labeled datasets. Therefore, we devise a novel *self-training framework* that exploits the *inverse functional relationships* between operators (e.g., the inverse of "`stack`" is known as "`unstack`"), to automatically build large amounts of training data without requiring humans to label, as we will explain in Figure 6. Briefly, in order to build a training example for operator O (e.g., "`stack`"), we start from a relational table R and apply the inverse of O, denoted by O^{-1} (e.g., "`unstack`"), to generate a table $T = O^{-1}(R)$, which we know is non-relational. For our task, given T as input, we know O must be its ground-truth transformation, because by definition $O(T) = O(O^{-1}(R)) = R$, which turns T back to its relational form R. This makes (T, O) an (example, label) pair that we can automatically generate at scale, and use as our training data.

Leveraging training data so generated, we develop an Auto-Tables system that can "learn-to-synthesize" table restructuring transformations, using a deep tabular model we develop inspired by CNN-like architectures popular in the computer vision literature. We show our approach is effective on real-world tasks, which can solve over 70% of test cases collected from user forums and spreadsheets, while being interactive with sub-second latency.

2. RELATED WORK

By-example transformation using program synthesis. There is a large body of prior work on using input/output examples to synthesize transformations. One class of techniques focuses on the so-called "row-to-row" transformations where one input row maps to one output row (e.g., TDE [31] and FlashFill [29]), which are orthogonal to the table-restructuring transformations in Auto-Tables, because these systems do not consider table restructuring operators (Table 1). Other forms of row-to-row transformations using partial specifications (e.g., transform-by-pattern [23, 48], transform-by-target [33, 34], and transform-for-joins [39, 51]), are also orthogonal to the problem we study for the same reason.

A second class of by-example transformation consider "table-to-table" operators, such as Foofah [32] and SQL-by-example techniques like PATSQL [44], QBO [45], and Scythe [46]. These techniques consider a subset of table-restructuring operators, which fall short in the Auto-Tables task as we will show experimentally. It is also worth highlighting, that unlike Auto-Tables that *takes no examples*, these prior systems require users to provide *an example output table*, which is a significant amount of effort required from users.

Computer vision models for object detection. Substantial progress has been made in the computer vision literature on object detection, with variants of CNN architectures being developed to extract salient visual features from pictures [43, 30, 35]. Given the "visual" nature of our problem shown in Figure 1, and the strong parallel between "pixles" in images and "rows/columns" in tables, both of which form two-dimensional rectangles, our model architecture is inspired by CNN-architectures for object detection, but specifically designed for our table transformation task.

3. PRELIMINARY AND PROBLEM

In this section, we will introduce table-restructuring operators, and describe our synthesis problem.

3.1 Table-restructuring operators

We consider 8 table-restructuring operators in our DSL, which are listed in Table 1. Based on our analysis of tables in the wild (in user spreadsheets and on the web), these operators cover a majority of scenarios required to relationalize tables. Note that since our synthesis framework uses self-supervision for training that is not tied to the specific choices of operators, our approach can be easily extended to include additional operators for new functionalities.

In this section, we will introduce the first 4 operators and their parameters shown in Table 1 (we will give additional details in our technical report [1] in the interest of space).

Stack. `Stack` is a Pandas operator [16] (also known as `melt` and `unpivot` in other contexts), that collapses contiguous blocks of homogeneous columns into two new columns. Like shown in Figure 1(a), column headers of the homogeneous columns ("`19-Oct`", "`20-Oct`", etc.) are converted into values of a new column called "`Date`", making it substantially easier to query (e.g., to filter using a range-predicate on "`Date`").

Parameters. In order to properly invoke `stack`, one needs to provide two important parameters, `start_idx` and `end_idx` (listed in the third column of Table 1), which specify the starting and ending column index of the homogeneous column-group that needs to be collapsed. In the case of Figure 1(a), we should use `start_idx`=3 (corresponding to column D) and `end_idx`=12 (column M).

Note that because in Auto-Tables we aim to synthesize complete transformation steps that can execute on input tables, which requires us to predict not only the operators (e.g., `stack` for the table in Figure 1(a)), but also the exact parameters values correctly (e.g., slightly different parameters such as `start_idx`=4 and `end_idx`=12 would fail to produce the desired transformation).

Wide-to-long. `Wide-to-long` is an operator in Pandas [18], that collapses repeating column groups into rows (similar functionality can also be found in R [20]). Figure 1(b) shows such an example, where "`Revenue/Units Sold/Margin`" from different years form column-groups that repeat once every 3 columns. All these repeating column-groups can collapse into 3 columns, with an additional "`Year`" column for year info from the original column headers, as shown on the right in Figure 1(b). Observe that `wide-to-long` is similar in spirit to `stack` (as both collapse homogeneous columns), although `stack` cannot produce the desired outcome when columns are repeating in groups as in this example.

Parameters. `Wide-to-long` has 3 parameters, in which `start_idx` and `end_idx` are similar to the ones used in `stack`. It has an additional parameter called "`delim`", which is the delimitor used to split the original column headers, to produce new column headers and data-values. For example, in the case of Figure 1(b), "`delim`" should be specified as "` -`" to produce: (1) a first part corresponding to values for the new "`Year`" column ("`2018`", "`2019`", etc.); and (2) a second part corresponding to the new column headers in the transformed table ("`Revenue`", "`Units Sold`", etc.).

Transpose. `Transpose` is a table-restructuring operator that converts rows to columns and columns to rows, and is used in other contexts such as in matrix computation. Figure 1(c) shows an example input table on the left, for which `transpose` is needed to produce the output table shown on the right, which would become relational and easy to query.

Parameters. Invoking `transpose` requires no parameters, as all rows and columns will be transposed.

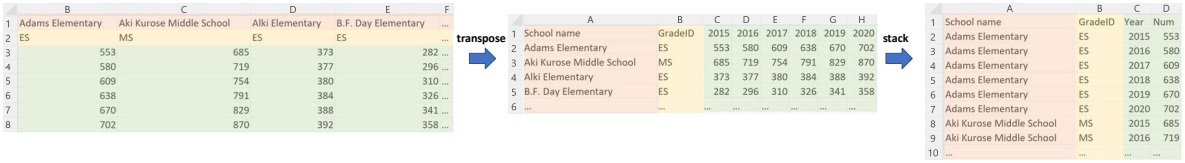


Figure 4: An example input table (on the left) that requires two transformation steps to relationalize: (1) a "`transpose`" step to swap rows and columns, (2) a "`stack`" step to collapse homogeneous columns (C to H) into two new columns. The resulting output table (on the right) becomes substantially easier to query with SQL (e.g., to filter and aggregate).

Pivot. Like `transpose`, `pivot` also converts rows to columns, as the example in Figure 1(d) shows. However, in this case rows show repeating-groups (whereas in `wide-to-long` columns show repeating-groups), which need to be transformed into columns, like shown on the right of Figure 1(d).

Parameters. `Pivot` has one parameter, "`repeat_frequency`", which specifies the frequency at which the rows repeat in the input table. In the case of Figure 1(d), this parameter should be set to 4, as the color pattern of rows would suggest.

Details of additional operators in Table 1 can be found in our technical report [1].

3.2 Problem statement

Given these table-restructuring operators listed in Table 1, we now introduce our synthesis problem as follows.

Definition 1. Given an input table T, and a set of operators $\mathbf{O} = \{stack, transpose, pivot, \ldots\}$, where each operator $O \in \mathbf{O}$ has a parameter space $P(O)$. Synthesize a sequence of multi-step transformations $M = (O_1(p_1), O_2(p_2), \ldots, O_k(p_k))$, with $O_i \in \mathbf{O}$ and $p_i \in P(O_i)$ for all $i \in [k]$, such that applying each step $O_i(p_i) \in M$ successively on T produces a relationalized version of T.

Note that in our task, we need to predict both the operator O_i and its exact parameters p_i correctly, each step along the way. This is challenging as the search space is large and grows exponentially with the number of steps.

Example 1. Given the input table T shown on the left of Figure 4, the ground-truth transformation M to relationalize T has two-steps: M = (transpose(), stack(start_idx:"2015", end_idx:"2020")). Here the first step "`transpose`" swaps the rows with columns, and the second step "`stack`" collapses the homogeneous columns (between column "2015" and "2020"). Note that this is the only correct sequence of steps – reordering the two steps, or using slightly different parameters (e.g., start_idx="2016" instead of "2015"), will all lead to incorrect output, which makes the problem challenging.

Also note that although we show synthesized programs using our DSL syntax, the resulting programs can be easily translated into different target languages, such as Python Pandas or R, which can then be directly invoked.

4. Auto-Tables: Learn-to-Synthesize

We now describe our proposed Auto-Tables, which learns to synthesize transformations. We will start with an architecture overview, before describing individual components.

4.1 Architecture overview

We represent our overall architecture in Figure 5. The system operates in two modes, with the upper-half of the figure showing the offline training-time pipeline, and the lower-half showing the online inference-time steps.

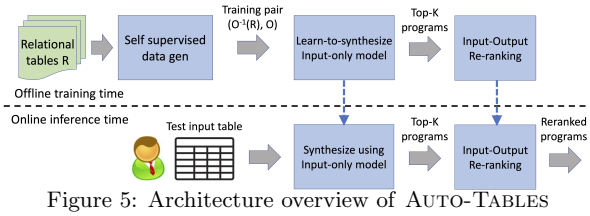


Figure 5: Architecture overview of Auto-Tables

At offline training time, Auto-Tables uses three main components: (1) A "training data generation" component that consumes large collections of relational tables R, to produce (example, label) pairs; (2) An "input-only synthesis" module that learns-to-synthesize using the training data, and (3) An "input-output re-ranking" module that considers both the input table and the output table (produced from the synthesized program), to find the most likely program.

The online inference-time part closely follows the offline steps, where we directly invoke the two models trained offline (the last two blue boxes shown in the figure). Given a user input table, we pass it through our input-only synthesis model, to identify top-k candidate programs, which are then re-ranked by the input-output model for final predictions.

We now describe these three modules in turn below.

4.2 Self-supervised training data generation

As discussed earlier, the examples in Figure 1 demonstrate that there are clear patterns in the input tables that we can exploit (e.g., repeating column-groups and row-groups) to predict required transformations for a given table. Note that these patterns are "visual" in nature, which can likely be captured by computer-vision-like algorithms.

The challenge however, is that unlike computer vision tasks that typically have large amounts of training data (e.g., ImageNet [26]) in the form of (image, label) pairs, in our synthesis task, there is no existing labeled data that we can leverage. Labeling tables manually from scratch are likely too expensive to scale.

Leverage inverse operators. To overcome the lack of data, we propose a novel self-supervision framework leveraging the inverse functional-relationships between operators, to automatically generate large amounts of training data without using humans labels.

Figure 6 shows the overall idea of this approach. For each operator O in our DSL that we want to learn-to-synthesize, we can find its inverse operator (or construct a sequence of steps that are functionally equivalent to its inverse), denoted by O^{-1}. For example, in the figure we can see that the inverse of "`transpose`" is "`transpose`", the inverse of "`stack`" is "`unstack`", while the inverse of "`wide-to-long`" can be constructed as a sequence of 3 steps ("`stack`" followed by "`split`" followed by "`pivot`").

The significance of the inverse operators, is that it allows

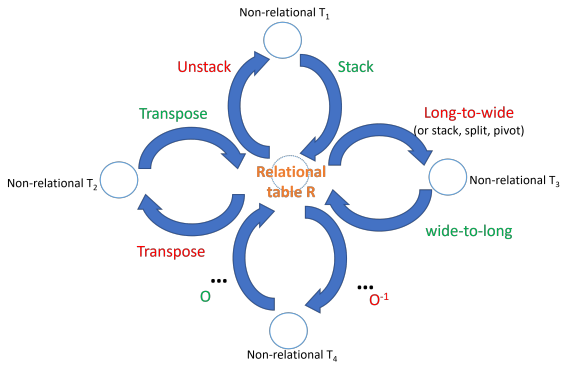


Figure 6: Leverage inverse operators to generate training data. In order to learn-to-synthesize operator O, we can start from any relational table R, apply its inverse operator O^{-1} to obtain $O^{-1}(R)$. Given $T = O^{-1}(R)$ as an input table, we know O must be its ground-truth transformation, because $O(O^{-1}(R)) = R$.

us to automatically generate training examples. Specifically, to build a training example for operator O (e.g., "`stack`"), we can sample any relational table R, and apply the inverse of O, or O^{-1} (e.g., "`unstack`"), to generate a non-relational table $T = O^{-1}(R)$. For our task, given T as input, we know O must be its ground-truth transformation, since by definition $O(T) = O(O^{-1}(R)) = R$, and R is known to be relational. This thus allows us to generate (T, O) as an (example, label) pair, which can be used for training.

Furthermore, we can easily produce such training examples at scale, by sampling: (1) different relational tables R; (2) different operators O; and (3) different parameters associated with each O, therefore addressing our lack of data problem in Auto-Tables.

Data Augmentation. Data augmentation [42] is a popular technique to enhance training data and improve model robustness. For example, in computer vision, it is observed that training using additional data generated from randomly flipped/rotated/cropped images, can lead to improved model performance (because an image that contains an object, say dog, should still contain the same object after it is flipped/rotated, etc.) [42].

In the same spirit, we augment each of our relational table R by (1) Cropping, or sampling contiguous blocks of rows and columns in R to produce a new table R'; and (2) Shuffling, or randomly reordering the rows/columns in R to create a new R'. In Auto-Tables, we start from over 15K relational tables crawled from public sources [37] (Section 5), and create around 20 augmented tables for each relational table R. This improves data diversity and end-to-end model performance, as we observe in our experiments.

4.3 Input-only Synthesis

After obtaining large amounts of training data in the form of (T, O_p) using self-supervision, we now describe our "input-only" model that takes T as input, to predict a suitable transformation O_p.

4.3.1 *Model architecture*

We develop a computer-vision inspired model specifically designed for our task, which scans through rows and columns to extract salient tabular features, reminiscent of how computer-vision models extract features from image pixels.

Our model architecture in Figure 7 consists of four sets of layers: (1) table embedding, (2) dimension reduction, (3) feature extraction, and (4) output layers. We will give a brief sketch of each below (details can be found in [1]).

Table embedding layers. Given an input table T, the embedding layer encodes each cell in T into a vector, to obtain an initial representation of T for training. At a high level, for each cell we want to capture both (1) the "*semantic features*" (e.g., people-names vs. company-names), and (2) the "*syntactic feature*" (e.g., data-type, string-length, punctuation, etc.), because both semantic and syntactic features provide valuable signals in our task, e.g., in determining whether rows/columns are homogeneous or similar.

We use the pre-trained Sentence-BERT embedding [40] for semantic features, and encode each cell with 39 pre-defined syntactic attributes (data types, string lengths, punctuation, etc.) as syntactic features, which are then concatenated, as shown in the left half of Figure 7.

Dimension reduction layers. Since the initial representation from the pre-trained Sentence-BERT has a large number of dimensions (with information likely not needed for our task, which can slow down training and increase the risk of over-fitting), we add dimension-reduction layers using two convolution layers with 1×1 kernels, to reduce the dimensionality. Note that we explicitly use 1×1 kernels so that the trained weights are shared across all table-cells, to produce consistent representations after dimension reduction.

Feature extraction layers. We next have feature extraction layers that are reminiscent of CNN [36] but specifically design for our table task. Recall from Figure 1 that the key signals for our task are:

- (1) identify whether values in row or column-directions are "similar" enough to be "homogeneous" (e.g., Figure 1(b) vs. Figure 1(c));
- (2) identify whether entire rows or columns are "similar" enough to show repeating patterns (e.g., Figure 1(b) vs. Figure 1(d)).

Intuitively, if we were to hand-write heuristics, then signal (1) above can be extracted by comparing the representations of adjacent cells in row- and column-directions. On the other hand, signal (2) can be extracted by computing the average representations of each row and column, which can then be used to find repeating patterns.

Based on this intuition, and given the strong parallel between the row/columns in tables and pixels in images, we design feature-extraction layers inspired by *convolution filters* [36] that are popular in CNN architectures to extract visual features from images [35, 43]. Specifically, as shown in Figure 7, we use 1x2 and 1x1 convolution filters followed by average-pooling, in both row- and column-directions, to represent rows/columns/header. Unlike general nxm filter used for image tasks (e.g., 3x3 and 5x5 filters in VGG [43] and ResNet [30]), our design of filters are tailored to our table task, because:

- (a) 1x2 filters can easily learn-to-compute signal (1) above (e.g., 1x2 filters with +1/-1 weights can identify the representation differences between neighboring cells, which when averaged, can identify homogeneity in rows/columns).
- (b) 1x1 filters can easily learn-to-compute signal (2) above (e.g., 1x1 filters with +1 weights followed by pooling, correspond to representations of entire rows/columns, which can be used to find repeating patterns in later layers).

We give a more detailed explanation and a concrete example in [1] to illustrate the design here.

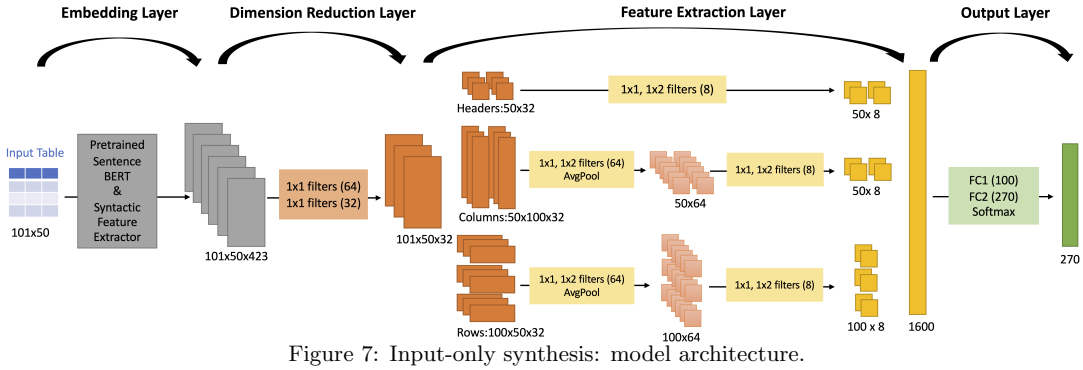


Figure 7: Input-only synthesis: model architecture.

Output layers. Our output layers use two fully connected layers followed by softmax classification, as shown in Figure 7, which produces an output vector that encodes both the predicted operator-type, and its parameters.

4.3.2 *Training and inference*

We now describe how we train the model, and perform inference to synthesize transformations.

Training time: Loss Function. Given a training input table T, its ground truth operator O and corresponding parameters $P = (p_1, p_2, ...)$, let $\hat{O}$ and $\hat{P} = (\hat{p}_1, \hat{p}_2, ...)$ be the model predicted probability distributions of O and P respectively. The training loss on T can be computed as the sum of loss on all predictions (both the operator-type, and parameters relevant to this operator):

$$Loss(T) = L(O, \hat{O}) + \sum_{p_i \in P, \hat{p}_i \in \hat{P}} L(p_i, \hat{p}_i) \quad (1)$$

Where $L(y, \hat{y})$ denotes the cross-entropy loss [38] commonly used in classification. Given large amounts of training data $\mathbf{T}$ (generated from our self-supervision in Section 4.2), we train our model by minimizing the overall training loss $\sum_{T \in \mathbf{T}} Loss(T)$ using gradient descent until convergence. We will refer to this trained model as H.

Inference time: Synthesizing transformations. At inference time, given an input T, our model H produces a probability for any candidate step O_P that is instantiated with operator O and parameters $P = (p_1, p_2, \ldots)$, denoted by $Pr(O_P|T)$:

$$Pr(O_P|T) = Pr(O) \cdot \prod_{p_i \in P} Pr(p_i) \quad (2)$$

Using the predicted probabilities, finding the most likely transformation step O^*_P given T is then simply:

$$O^*_P = \arg\max_{O,P} Pr(O_P|T) \quad (3)$$

This gives us the most likely one-step transformation given T. As we showed in Figure 4, certain tables may require multiple transformation steps for our task.

To synthesize multi-step transformations, intuitively we can invoke our predictions step-by-step until no suitable transformation can be found. Specifically, given an input table T, at step (1) we can find the most likely transformation O^1_P for T using Equation (3), such that we can apply O^1_P on T to produce an output table $O^1_P(T)$. We then iterate, and at step (2) we feed $O^1_P(T)$ as the new input table into our model, to predict the most likely $O^2_P(T)$, and produce an output table $O^2_P(O^1_P(T))$. This iterates until at the k-th step, a "`none`" transformation is predicted (recall that "`none`" is a no-op operator in our DSL in Table 1, to indicate that the input table is already relational and requires no transformations). The resulting $M = (O^1_P, O^2_P, \ldots)$ then becomes the multi-step transformations we synthesize for T.

We defer details of our inference-time algorithm, as well our last component, "input/output re-ranking", to [1].

5. EXPERIMENTS

We perform extensive evaluation on the performance of different algorithms using real test data. Our labeled benchmark data is available on GitHub[1] for future research.

5.1 Experimental Setup

Benchmarks. To study the performance of our method in real-world scenarios, we compile an ATBench benchmark using real cases from three sources:

(1) Online user forums. We sample 23 questions from StackOverflow and Excel user forums, where users ask questions about table restructuring, with sample input/output tables to demonstrate their needs (e.g., Figure 3).

(2) Jupyter notebooks. We sample 79 table-restructuring steps performed by data scientists, extracted from the Jupyter Notebooks (crawled from [47, 48]). We use the transformations programmed by data scientists as the ground truth.

(3) Excel spreadsheets + Web tables. Tables "in the wild" often require transformations before they are fit for analysis (e.g., Figure 1 and 2). We sample 56 such web-tables and 86 spreadsheet-tables, and manually write the desired transformations as the ground truth.

Combining these sources, we get a total of 244 test cases as our ATBench (of which 26 cases require multi-step transformations). Each test case consists of an input table T, ground-truth transformations M_g, and an output table $M_g(T)$ that is relational.

Detailed statistics of the benchmark can be found in [1].

Evaluation Metrics. We evaluate the quality and efficiency of different algorithms in synthesizing transformations.

Quality. Given an input table T, an algorithm A may generate top-k transformations $(\hat{M}_1, \hat{M}_2, ...\hat{M}_k)$, ranked by probabilities, for users to inspect and pick. We evaluate synthesis quality using the standard $Hit@k$ metric [41]:

$$Hit@k(T) = \sum_{i=1}^{k} \mathbf{1}(\hat{M}_i(T) = M_g(T))$$

which looks for exact matches between the top-k ranked predictions $(\hat{M}_i(T), 1 \le i \le k)$ and the ground-truth $M_g(T)$.

[1] `https://github.com/LiPengCS/Auto-Tables-Benchmark`

Table 2: Quality comparison using Hit@k, on 244 test cases

Method	No-example methods				By-example methods			
	Auto-Tables	TaBERT	TURL	GPT-3.5-fs	FF	FR	SQ	SC
Hit @ 1	**0.570**	0.193	0.029	0.196	0.283	0.336	0	0
Hit @ 2	**0.697**	0.455	0.071	-	-	-	0	0
Hit @ 3	**0.75**	0.545	0.109	-	-	-	0	0
Upper-bound	-	-	-	-	0.471	0.545	0.369	0.369

Table 3: Synthesis latency per test case

Method	Auto-Tables	Foofah (excl. 110 timeout cases)	FlashRelate (excl. 91 timeout cases)
50 %tile	**0.127s**	0.287s + human effort	3.4s + human effort
90 %tile	**0.511s**	22.891s + human effort	57.16s + human effort
95 %tile	**0.685s**	39.188s + human effort	348.6s + human effort
Average	**0.224s**	5.996s + human effort	59.194s + human effort

The overall $Hit@k$ on the entire benchmark, is simply the average across all test cases. We report $Hit@k$ up to $k = 3$.

Methods. We experiment using the following methods.

- Auto-Tables. This is our approach, and unlike other prior work, is the only one that does not require users to provide input/output examples. To train Auto-Tables, We use 15K base relational tables (extracted from Power-BI models crawled from public sources [37]), to generate 1.4M (input-table, transformation) pairs evenly distributed across 8 operators, following the self-supervision procedure in Section 4.2.
- *Foofah (FF)* [32] synthesizes transformations based on input/output examples. We use 100 output cells from the ground-truth output table for Foofah to synthesize programs, using the authors original implementation [10].
- *Flash-Relate (FR)* [24] is another approach to synthesize table-level transformations by examples. We also use 100 example output cells from the ground-truth to synthesize transformations, using an open-source re-implementation of FlashRelate [21].
- *SQLSynthesizer (SQ)* [50] is a SQL-by-example algorithm that synthesizes SQL queries based on input/output examples. We use the authors implementation [22], provide it with 100 example output cells.
- *Scythe (SC)* [46] is another SQL-by-example method. We used the author's implementation [19] and provide it with 100 example output cells, like previous methods.
- *TaBERT* [49] is a pre-trained table representation. we replace the table representation in Auto-Tables (i.e., output of the feature extraction layer in Figure 7) with TaBERT's representation, and train the following fully connected layers using the same training data as ours.
- *TURL* [27] is another table representation approach for data integration tasks. Similar to *TaBERT*, we test the effectiveness of TURL by replacing Auto-Tables representation with TURL's.
- *GPT* [25] We perform few-shot in-context learning on GPT-3.5 ("gpt-3.5-turbo", accessed in July 2023) as a baseline. We provide one example per operator, for a total of 7 examples in our few-shot prompt.

5.2 Experiment Results

Quality Comparison. Table 2 shows the comparison between Auto-Tables and baselines, evaluated on our benchmark with 244 test cases. We group all methods into two classes: (1) "No-example methods" that do not require users to provide any input/output examples, which include our Auto-Tables, and variants of Auto-Tables that use TaBERT and TURL for table representations, respectively; and (2) "By-example methods" that include Foofah (FF), FlashRelate (FR), SQLSynthesizer (SQ), and Scythe (SC), all of which are provided with 100 ground truth example cells.

As we can see, Auto-Tables significantly outperforms all other methods, successfully transforming 75% of test cases in its top-3, *without needing users to provide any examples*, despite the challenging nature of our tasks. Recall that in our task, even for a single-step transformation, there are thousands of possible operators+parameters to choose from (e.g., a table with 50 columns that requires "`stack`" will have 50x50 = 2,500 possible parameters of start_idx and end_idx) and for two-step transformations, the search space is in the millions (e.g., for "`stack`" alone it is $2500^2 \approx 6M$), which is clearly non-trivial.

Compared to other no-example methods, Auto-Tables outperforms TaBERT and TURL respectively by 37.7 and 54.1 percentage point on Hit@1, 20.5 and 64.1 percentage point on Hit@3. This shows the strong benefits for using our proposed table representation and model architecture, which are specifically designed for the table transformation task (Section 4.3).

Compared to by-example methods, the improvement of Auto-Tables is similarly strong. Considering the fact that these baselines use 100 output example cells (which users need to manually type), whereas our method uses 0 examples, we argue that Auto-Tables is clearly a better fit for the table-restructuring task at hand. Since some of these methods (FF and FR) only return top-1 programs, we also report in the last row their "upper-bound" coverage, based on their DSL (assuming all transformations supported in their DSL can be successfully synthesized).

Additional quality results. We report additional results on quality, such as a breakdown by benchmark sources, and Hit@K in the presence of input tables that are already relational (for which Auto-Tables should correctly detect and not over-trigger, by performing no transformations), in our technical report [1].

Running Time. Table 3 compares the average and 50/90/95-th percentile latency, of all methods to synthesize one test case. Auto-Tables is interactive with sub-second latency on almost all cases, whose average is 0.224. Foofah and FlashRelate take considerably longer to synthesize, even after we exclude cases that time-out after 30 minutes. This is also not counting the time that users would have to spend typing in output examples for these by-example methods, which we believe make Auto-Tables substantially more user-friendly for our transformation task.

We report additional results, such as ablation, sensitivity and error analysis, in our technical report [1].

6. CONCLUSIONS AND FUTURE WORK

We propose a new paradigm to synthesize relationalization transformations without examples, obviating the need for users to provide input/output examples, which is a substantial departure from prior work. Future directions include extending the functionality to a broader set of operators, and exploring the applicability of this technique on other classes of transformations.

Acknowledgments. We thank Dr. Kexin Rong and Dr. Xu Chu for their generous support and valuable feedback, as well as three anonymous VLDB reviewers for their helpful comments on our manuscript.

7. REFERENCES

[1] Auto-Tables: full version. https://arxiv.org/abs/2307.14565.

[2] Example Excel forum question: Hard to query without transformations (Retrieved in 02/2023). https://techcommunity.microsoft.com/t5/excel/power-query-data-cleaning-unpivot-transpose-etc/m-p/2400300.

[3] Example Excel forum question: Hard to query without transformations (Retrieved in 02/2023). https://techcommunity.microsoft.com/t5/excel/unpivot-grouped-data/m-p/3686239.

[4] Example Excel forum question: Hard to query without transformations (Retrieved in 02/2023). https://techcommunity.microsoft.com/t5/excel/unpivot-monthly-data/m-p/1867836.

[5] Example Excel forum question: Table analysis provides unexpected results (Retrieved in 02/2023). https://answers.microsoft.com/en-us/msoffice/forum/all/excel-ideas-feature/c9574cf9-dccc-4356-95d3-07d268e39d82.

[6] Example Excel forum question to relationalize tables: Data restructuring using Excel (Retrieved in 02/2023). https://techcommunity.microsoft.com/t5/excel/data-restructuring-using-excel/m-p/287547.

[7] Example Excel forum question to relationalize tables: Pivot chart 4 columns (Retrieved in 02/2023). https://techcommunity.microsoft.com/t5/excel/pivot-chart-4-columns-set-responses-to-4-questions/m-p/2329880.

[8] Example Excel forum question to relationalize tables: Pivot table issue. (Retrieved in 02/2023). https://techcommunity.microsoft.com/t5/excel/pivot-table-issue/m-p/3015448.

[9] Example Excel forum question to relationalize tables: Transpose data for analysis (Retrieved in 02/2023). https://techcommunity.microsoft.com/t5/excel/transposing-data-for-better-analysis/m-p/1297106.

[10] Foofah code on GitHub. https://github.com/umich-dbgroup/foofah.

[11] Pandas API in Python. https://pandas.pydata.org/.

[12] Pandas operator: Explode. (Retrieved in 02/2023). https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.explode.html.

[13] Pandas operator: FFill. (Retrieved in 02/2023). https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.ffill.html.

[14] Pandas operator: Melt. (Retrieved in 02/2023). https://pandas.pydata.org/docs/reference/api/pandas.melt.html.

[15] Pandas operator: Pivot. (Retrieved in 02/2023). https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.pivot.html.

[16] Pandas operator: Stack. (Retrieved in 02/2023). https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.stack.html.

[17] Pandas operator: Transpose. (Retrieved in 02/2023). https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.transpose.html.

[18] Pandas operator: Wide-to-long. (Retrieved in 02/2023). https://pandas.pydata.org/docs/reference/api/pandas.wide_to_long.html.

[19] PATSQL code on GitHub. https://github.com/NAIST-SE/PATSQL.

[20] R operator: pivot-longer, which is similar to Wide-to-long. (Retrieved in 02/2023). https://tidyr.tidyverse.org/reference/pivot_longer.html.

[21] Reimplementation of FlashRelate code on GitHub. https://github.com/BEE-Synth/Bee/tree/291a824622e36fccfa43461e85be3f836e3f4eff/Eval/Benchmarks/Spreadsheet/flashrelate-01.

[22] Scythe code on GitHub. https://github.com/Mestway/Scythe.

[23] Trifacta: Standardize Using Patterns. (Retrieved in 07/2023). https://docs.trifacta.com/display/DP/Standardize+Using+Patterns.

[24] D. W. Barowy, S. Gulwani, T. Hart, and B. Zorn. Flashrelate: extracting relational data from semi-structured spreadsheets using examples. *ACM SIGPLAN Notices*, 50(6):218–228, 2015.

[25] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[26] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[27] X. Deng, H. Sun, A. Lees, Y. Wu, and C. Yu. Turl: Table understanding through representation learning. *ACM SIGMOD Record*, 51(1):33–40, 2022.

[28] Y. Gao, S. Huang, and A. Parameswaran. Navigating the data lake with datamaran: Automatically extracting structure from log datasets. In *Proceedings of the 2018 International Conference on Management of Data*, pages 943–958, 2018.

[29] S. Gulwani, W. R. Harris, and R. Singh. Spreadsheet data manipulation using examples. *Communications of the ACM*, 55(8):97–105, 2012.

[30] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[31] Y. He, X. Chu, K. Ganjam, Y. Zheng, V. Narasayya, and S. Chaudhuri. Transform-data-by-example (tde) an extensible search engine for data transformations. *Proceedings of the VLDB Endowment*, 11(10):1165–1177, 2018.

[32] Z. Jin, M. R. Anderson, M. Cafarella, and H. Jagadish. Foofah: Transforming data by example. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 683–698, 2017.

[33] Z. Jin, Y. He, and S. Chauduri. Auto-transform: learning-to-transform by patterns. *Proceedings of the VLDB Endowment*, 13(12):2368–2381, 2020.

[34] M. Koehler, E. Abel, A. Bogatu, C. Civili, L. Mazilu, N. Konstantinou, A. A. Fernandes, J. Keane, L. Libkin, and N. W. Paton. Incorporating data context to cost-effectively automate end-to-end data wrangling. *IEEE Transactions on Big Data*, 7(1):169–186, 2019.

[35] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. *CACM*, 60(6):84–90, 2017.

[36] Z. Li, F. Liu, W. Yang, S. Peng, and J. Zhou. A survey of convolutional neural networks: analysis, applications, and prospects. *IEEE transactions on neural networks and learning systems*, 2021.

[37] Y. Lin, Y. He, and S. Chaudhuri. Auto-bi: Automatically build bi-models leveraging local join prediction and global schema graph. *Proceedings of the VLDB Endowment*, 2023.

[38] K. P. Murphy. *Machine learning: a probabilistic perspective*. MIT press, 2012.

[39] A. D. Nobari and D. Rafiei. Efficiently transforming tables for joinability. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, pages 1649–1661. IEEE, 2022.

[40] N. Reimers and I. Gurevych. Sentence-bert: Sentence embeddings using siamese bert-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 11 2019.

[41] H. Schütze, C. D. Manning, and P. Raghavan. *Introduction to information retrieval*, volume 39. Cambridge University Press Cambridge, 2008.

[42] C. Shorten and T. M. Khoshgoftaar. A survey on image data augmentation for deep learning. *Journal of big data*, 6(1):1–48, 2019.

[43] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[44] K. Takenouchi, T. Ishio, J. Okada, and Y. Sakata. Patsql: efficient synthesis of sql queries from example tables with quick inference of projected columns. *arXiv preprint arXiv:2010.05807*, 2020.

[45] Q. T. Tran, C.-Y. Chan, and S. Parthasarathy. Query by output. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 535–548, 2009.

[46] C. Wang, A. Cheung, and R. Bodik. Synthesizing highly expressive sql queries from input-output examples. In *SIGPLAN*, pages 452–466, 2017.

[47] C. Yan and Y. He. Auto-suggest: Learning-to-recommend data preparation steps using data science notebooks. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 1539–1554, 2020.

[48] J. Yang, Y. He, and S. Chaudhuri. Auto-pipeline: synthesizing complex data pipelines by-target using reinforcement learning and search. *Proceedings of the VLDB Endowment*, 2021.

[49] P. Yin, G. Neubig, W.-t. Yih, and S. Riedel. Tabert: Pretraining for joint understanding of textual and tabular data. *arXiv preprint arXiv:2005.08314*, 2020.

[50] S. Zhang and Y. Sun. Automatically synthesizing sql queries from input-output examples. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 224–234. IEEE, 2013.

[51] E. Zhu, Y. He, and S. Chaudhuri. Auto-join: Joining tables by leveraging transformations. *Proceedings of the VLDB Endowment*, 10(10):1034–1045, 2017.

Technical Perspective: A Fresh Look at Stream Computation through DSP Glasses

Dan Olteanu
Department of Informatics, University of Zurich
dan.olteanu@uzh.ch

DBSP (Data Base Stream Processing) is a *simple yet expressive language for stream computation* that draws inspiration from DSP (Digital Signal Processing). In DBSP, stream computation is expressed using circuits of stream operators whose input and output are (possibly infinite) sequences of database updates.

Queries in languages such as SQL and even Datalog with recursion can be compiled into DBSP circuits in a modular way, where logical relational algebra operators such as projection, selection, join, union, and difference are compiled into stream operators that are composed into a circuit implementing the logic of the query. Special stream operators are also used to delay streams, to gather the entire stream into a database, and to generate the sequence of changes between any two subsequent stream elements.

DBSP's stream circuits are a powerful artifact. They are an *intermediate representation* between the high-level declarative queries and the low-level highly efficient code. They can be optimized by merging operators and by transforming them using rewrite rules. A notable optimization is for the recursive Datalog program that computes reachability in a graph. The circuit obtained for this program implements the classical naïve computation of reachability: Each iteration adds the pairs of start and end nodes of increasingly longer paths. Its optimization naturally recovers the classical semi-naïve evaluation: Now each iteration only adds those pairs of start and end nodes of paths that do not have shorter paths between them, so these pairs were not already discovered in earlier iterations.

Most importantly for stream computation, the circuits can be *automatically incrementalized*: For DBSP, there is a simple and elegant procedure that turns any circuit that computes a query over a database into a circuit that incrementally maintains the query under the stream of changes to the database. At its core, this procedure uses the property that a composite query can be incrementalized by incrementalizing each of its subqueries independently. A prime example is the automatic incrementalization of the equality join operator. The DBSP framework recovers the well-known delta rule for a join: Given the join of two relations and changes to both relations, the change to the join result is the union of (i) the join of the changes; (ii) the join of the first relation and of the change to the second relation; and (iii) the join of the second relation and of the change to the first relation. Query incrementalization, which lies at the foundation of *incremental view maintenance*, is proved formally in the context of DBSP using a theorem prover.

A further key ingredient of the DBSP framework is the use of $\mathbb{Z}$-sets to keep track in a *uniform way* of the nature and amount of changes: Changes consist of database tuples annotated with multiplicities. Inserts are expressed using tuples with positive multiplicities, whereas deletes are expressed using tuples with negative multiplicities. Allowing negative multiplicities in the database offers great flexibility when dealing with out-of-order updates. To see this, consider an insert and a delete of the same tuple. Regardless of the order of their arrival, the DBSP framework concludes that the two updates cancel each other. Virtually all commercial database systems, which provide support for incremental view maintenance, follow a non-confluent update semantics as they obtain different outcomes depending on the order of the two updates: the tuple is (not) in the database if the delete (insert) comes first. The latter update semantics can also be recovered by DBSP using a *distinct* operator in DBSP circuits. Using tuple multiplicities for incremental view maintenance goes back to the counting algorithm by Gupta et al from 1993. These days, $\mathbb{Z}$-sets are used critically in research prototypes such as DBToaster, F-IVM, and Crown and in the RelationalAI commercial engine. It is surprising that despite the clean update semantics and rather small implementation overhead of $\mathbb{Z}$-sets, so far their use remains limited.

Overall, DBSP is a fresh look at the long-standing problem of maintaining the result of relational queries under updates to the input database. Thanks to its unifying treatment of both non-recursive and recursive queries, DBSP follows in the footsteps of Differential Dataflow, a much celebrated framework that caters for general stream computation. The intermediate representation remains however the key feature of DBSP: High-level query languages can be compiled into circuits, which are more fine-grained than queries and allow for lower-level optimization and generation of efficiently executable code. It would be exciting to investigate how recent maintenance strategies, which achieve (amortized) constant time per single-tuple update for several classes of non-recursive queries, can be adapted to the DBSP framework. This would require extensions with stream operators that maintain state in the form of auxiliary data structures, with worst-case optimal join algorithms, and with factorized join computation and maintenance.

DBSP: Incremental Computation on Streams and Its Applications to Databases

Mihai Budiu
Feldera
mbudiu@feldera.com

Tej Chajed
Univ. of Wisconsin-Madison
chajed@wisc.edu

Frank McSherry
Materialize Inc.
mcsherry@materialize.com

Leonid Ryzhyk
Feldera
leonid@feldera.com

Val Tannen
University of Pennsylvania
val@seas.upenn.edu

ABSTRACT

We describe DBSP, a framework for incremental computation. Incremental computations repeatedly evaluate a function on some input values that are "changing". The goal of an efficient implementation is to "reuse" previously computed results. Ideally, when presented with a new change to the input, an incremental computation should only perform work proportional to the size of the changes of the input, rather than to the size of the entire dataset.

In databases "incremental computation" is known as Incremental View Maintenance (IVM); IVM has long been a central problem of database theory and practice. Many solutions have been proposed for restricted classes of computation or of changes, but we are seeking a general solution.

We start by defining incremental computations as computations on data streams, i.e., sequences of data values, by borrowing ideas from Digital Signal Processing.

Using these tools, we give a general solution to the incremental computation problem in 4 steps: (1) we describe a simple but expressive language called DBSP for describing computations over data streams; (2) we give a new mathematical definition of incremental computation for DBSP programs; (3) we give a general algorithm for converting any DBSP program into an incremental program. The algorithm reduces the problem of incrementalizing a complex query to the problem of incrementalizing the primitive operations that compose the query. Finally, (4) we show that practical database query languages, such as SQL and Datalog, can be directly implemented on top of DBSP, using primitives with efficient incremental implementations. As a consequence, we obtain a general recipe for efficient IVM for essentially arbitrary queries written in all these languages.

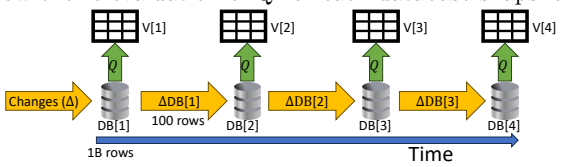
1. INTRODUCTION

1.1 Incremental computation

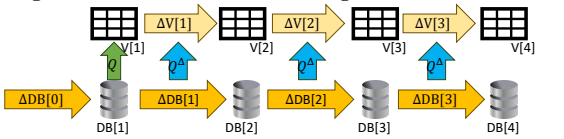
Incremental view maintenance (IVM) is an important and well-studied problem in databases [14]. The IVM problem can be stated as follows: we are given a large database DB (say 1 billion records) and a view V, described by a query Q. The goal of IVM is to keep the contents of V up-to-date in response to changes of the database.

As a concrete example, consider the following view definition statement in SQL: `CREATE VIEW V AS SELECT * FROM T WHERE Age >= 10`. In this example the query Q defining the view V is `SELECT * FROM T WHERE Age >= 10`. The view `V` always contains all the rows of table `T` whose value for the column `Age` is greater than or equal to 10.

In general a query is a function applied to the database state: $V = Q(DB)$. A naïve solution re-executes query Q every time the database changes, illustrated in the following diagram. Time is the horizontal axis; the horizontal arrows labeled with Δ depict changes to the database, which we assume are much smaller than the database itself (e.g., a change could touch perhaps 100 records). The "up" arrows show the re-evaluation of Q for each database snapshot.



The naive solution is expensive. After the first version of the view has been constructed, an ideal algorithm would compute only *changes* to the view ΔV doing work $O(|\Delta DB|)$. Ideally, we want to construct a new query Q^Δ with the property that $\Delta V = Q^\Delta(\Delta DB)$, i.e., Q^Δ can compute the change of the view from the change of the database:



We call Q^Δ the *incremental* version of Q. If one thinks of Q^Δ as a function of ΔDB, one can show that the ideal solution as described above is impossible to reach.

In this paper we propose a new way to define Q^Δ, as a form of *computation on streams*. Our model is inspired by Digital Signal Processing DSP [16], applied to databases, hence the name DBSP. Q^Δ can be very efficient. As for traditional database queries, the performance of Q^Δ depends both on the query Q but also on the actual data that the query is applied to. Informally, Q^Δ built by our algorithm, is faster

 This is a simplified version of the paper entitled DBSP: Automatic Incremental View Maintenance for Rich Query Languages, published in PVLDB, Vol. 16, Issue 7, ISSN 2150-8097 [5]. DOI: https://doi.org/10.14778/3587136.3587137

than Q by a factor of $O(|DB|/|\Delta DB|)$. In practice this may be an improvement of several orders of magnitude. For our example above $|DB| \approx 10^9$ and $|\Delta DB| \approx 10^2$, this can make Q^Δ **10 million times** faster!

Instead of treating the database as a large changing object, we model it as a *sequence* or *stream* of database snapshots. Similarly, consecutive view snapshots form a stream. DBSP is a simple programming language computing on streams; inputs and outputs are streams of arbitrary values.

The DBSP language has only 4 operators. However, it can express a rich set of computations on streams, including repeated computations (similar to the repeated queries Q above), recursive computations that compute fixed points (like Datalog programs), streaming computations, and incremental computations (which we define shortly). The full paper [5] gives a precise mathematical description of DBSP, this presentation is simplified to convey the main intuitions. We omit the related work section from this presentation.

The central result of this paper is Algorithm 4.1 which, given a DBSP program that computes on a stream of values, mechanically transforms it into an incremental DBSP program that computes on a stream of changes.

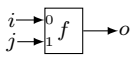
DBSP is not tied to databases in any way; it is in fact a Turing-complete language that can be used for many other purposes. But it works particularly well in the area of databases, for two reasons:

• DBSP operates on values from a commutative group. Databases can be modeled as a commutative group.

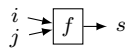
• DBSP reduces the problem of incrementalizing a complex program to the problem of incrementalizing each primitive operation that appears in the program. For databases there are known efficient incremental implementations for all primitive operations.

1.2 Circuits and Streams

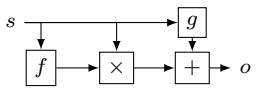
In this paper we use circuit diagrams to depict programs. In a circuit a rectangle represents a function, and an arrow represents an input or output value. The following diagram shows a function f consuming two inputs i (input 0) and j (input 1) and producing one output $o = f(i, j)$:



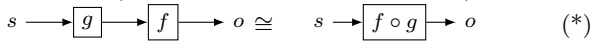
Most of the functions we deal with are commutative, so we can skip inputs label, displaying the circuit above as:



Functions, and their circuits, can be composed, as in the following example for the function $o = g(s) + (f(s) \times s)$:



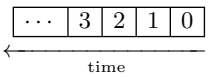
We say that two circuits are **equivalent** if they compute the same function. We use the symbol $\cong$ to indicate circuit equivalence. For example, we have the following circuit equivalence (where $\circ$ is function composition):



(*)

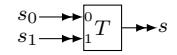
1.3 Streams

The core notion of DBSP is the **stream**. Given a set A, a **stream** *of values from* A is an infinite sequence of values from A. $\mathcal{S}_A$ denotes the set of all streams with values from A. We write $s[t]$ for the t-th element of the stream s. Think of t as the "time" and of $s[t] \in A$ as the value of the stream s "at time" t. We show streams as a sequence of boxes, with time from *right to left*: e.g., the stream $s[t] = t$ is:

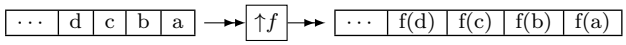


A **stream operator** is a function that computes on streams and produces streams. In general we use "operator" for streams, and "function" for computations on "scalar" values.

We use arrows with a double head to depict streams. The following diagram shows a stream operator T consuming two input streams s_0 and s_1, producing one output stream s:



We write $s = T(s_0, s_1)$. Given a function $f : A \to B$, we define a stream operator $\uparrow f : \mathcal{S}_A \to \mathcal{S}_B$ (read as "f lifted") by applying function f to each input value independently:



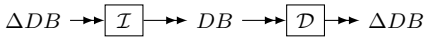
To simplify the notation, we write $a + b$ for streams a, b instead of $a(\uparrow+)b$; we also write $-a$ instead of $(\uparrow-)a$.

1.4 Databases as streams

We generally think of streams as sequences of "small" values, such as insertions or deletions in a database. However, we also treat the whole database as a *stream of database snapshots*. We model a database as a stream DB. Time is not wall-clock time, but counts the transactions applied to the database. Since transactions are linearizable, they have a total order. $DB[t]$ is the snapshot of the database contents after t transactions have been applied. This notation is apparent in the diagrams in Section 1.1.

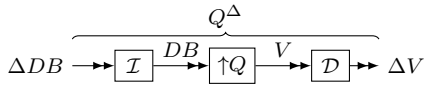
Database transactions also form a stream ΔDB, this time a stream of *changes*, or *deltas*, that are applied to the database. The values of this stream are defined by $(\Delta DB)[t] = DB[t] - DB[t-1]$, where "$-$" stands for the difference between two databases, a notion that we will soon make more precise. The ΔDB stream can be produced from the DB stream by the *stream differentiation* operator $\mathcal{D}$; this operator produces as its output the stream of changes from its input stream; we have thus $\mathcal{D}(DB) = \Delta DB$.

Conversely, the database snapshot at time t is the cumulative result of applying all transactions up to t: $DB[t] = \Delta DB[0] + \Delta DB[1] + \ldots + \Delta DB[t]$. The stream operator $\mathcal{I}$ is defined to produce each output by adding up all previous inputs. We call $\mathcal{I}$ *stream integration*, the inverse of differentiation. The following diagram shows the relationship between the streams ΔDB and DB:



A view in this model is also a stream. Suppose query Q defining a view V. For each snapshot of the database stream we have a snapshot of the view: $V[t] = Q(DB[t])$. A view is thus just a lifted query: $V = (\uparrow Q)(DB)$.

Armed with these basic definitions, we can precisely define IVM. What does it mean to maintain a view incrementally? A maintenance algorithm needs to compute the *changes* to the view given the changes to the database. Given a query Q, a key contribution of this paper is the definition of its *incremental version* Q^Δ, using stream integration and differentiation, depicted graphically as:



Mathematically: $Q^\Delta = \mathcal{D} \circ (\uparrow Q) \circ \mathcal{I}$. The incremental version of a query Q is a *streaming operator* Q^Δ which computes directly on changes and produces changes. The incremental version of a query is thus always well-defined. The above definition gives us one way to compute a query incrementally, but applying it naively produces an inefficient execution, since it reconstructs the database at each step. It is in fact as bad as the naive solution. In Section 3 we show how we can optimize the implementation of Q^Δ. The key property is that the we can "push" the $.^\Delta$ operator "down" in a query plan: $(Q_1 \circ Q_2)^\Delta = {Q_1}^\Delta \circ {Q_2}^\Delta$.

Armed with this general theory of incremental computation, in Section 4 we show how to model relational queries in DBSP. This immediately gives us a general algorithm to compute the incremental version of any relational query. These results were previously known, but they are cleanly modeled by DBSP. Section 5 shows how programs containing recursion can be implemented and incrementalized in DBSP. For example, given an implementation of transitive closure in the natural recursive way, our algorithm produces a program that efficiently maintains the transitive closure of a graph as nodes and edges are added and deleted.

1.5 Contributions

This work makes the following contributions:

(1) We introduce DBSP, a simple but expressive language for streaming computation. DBSP gives an elegant formal foundation unifying the manipulation of streaming and incremental computations.

(2) An algorithm for incrementalizing any streaming computation expressed in DBSP that handles arbitrary insertions and deletions from any of the data sources.

(3) An illustration of how DBSP can model various classes of practical queries, such as relational algebra, nested relations, aggregations, and Datalog.

(4) The first general and machine-checked theory of IVM. All the theoretical results in the original version of this paper [5] have been checked [7] using the Lean proof assistant [8].

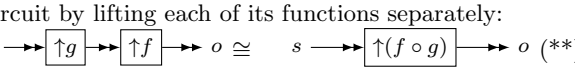
(5) A practical open-source implementation of this theory as a runtime and a SQL compiler.

2. STREAM OPERATORS

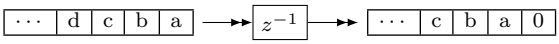
For the rest of this paper we require the set of values A of a stream $\mathcal{S}_A$ to form a commutative group, with operations $+$, $-$, and a 0 (zero) value. The *plus* defines what it means to *add* new data, while the *minus* allows us to compute differences (deltas). We show later that this requirement is not a problem for using DBSP in the context of databases.

Stream operators are very powerful mathematically, but in DBSP we restrict ourselves to a very small subset. All DBSP computations are *causal* [4]: the output at time t is produced immediately after all inputs up at time t have been received; the output at time t cannot depend on inputs arriving after t.

The following circuit equivalence tells us that we can lift a circuit by lifting each of its functions separately:

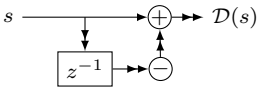


The **delay operator** z^{-1} produces an output stream by delaying its input by one step (and starting with a zero)[1]:

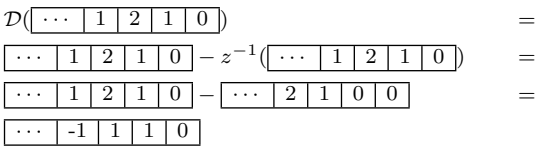


A very important property of the delay operator is that it produces the first output *before* having received the first input, and it produces the second output before having received the second input, etc.

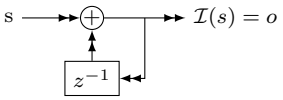
We define the **differentiation operator** as a composition of several other operators: $\mathcal{D}(s) \stackrel{\text{def}}{=} s - z^{-1}(s)$, shown as:



If s is a stream, then $\mathcal{D}(s)$ is the *stream of changes* of s; a value in the output is the difference between two consecutive values in the input. As an example:



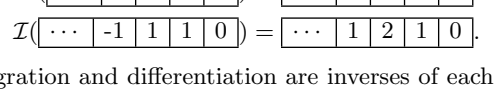
The **integration operator** is given by the following circuit:



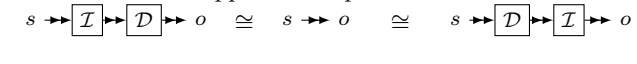
While this definition may seem strange, because the output stream is used to compute itself, the use of the delay in the "feedback" loop ensures that only *previous* values of the output are used in computing the current one. Using the notation $o = \mathcal{I}(s)$ to make formulas more readable, we can see the contents of stream o is produced step by step:

$$o[0] = s[0] + (z^{-1}(o))[0] = s[0] + 0 = s[0]$$
$$o[1] = s[1] + (z^{-1}(o))[1] = s[1] + o[0] = s[1] + s[0]$$
$$o[2] = s[2] + (z^{-1}(o))[2] = s[2] + o[1] = s[2] + (s[1] + s[0])$$

In general, $\mathcal{I}(s)[t] = o[t] = \sum_{i \le t} s[i]$. Examples:



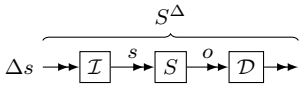
Integration and differentiation are inverses of each other: while $\mathcal{D}$ computes the changes of a stream, $\mathcal{I}$ reconstitutes the original stream given the stream of changes. $\mathcal{I}$ and $\mathcal{D}$ "cancel out" when applied in sequence:



3. INCREMENTAL VIEW MAINTENANCE

The results in this section are not specific to databases, they hold for any stream computations, but we hint about their applicability for databases.

Given a stream operator $S : \mathcal{S}_A \to \mathcal{S}_B$ we define the **incremental version** of S as:



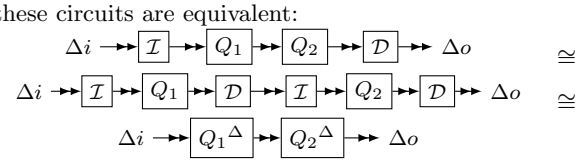
[1]This bizarre name comes from digital signal processing.

If S computes on a stream s, then S^Δ computes on a stream of changes to s. If S produces a stream o, then S^Δ produces the stream of changes to o. Note that this definition does not require S to be a lifted function.

For an operator with multiple inputs and outputs we define the incremental version by applying $\mathcal{I}$ to each input, and $\mathcal{D}$ to each output, e.g.: $T^\Delta(a,b) \stackrel{\text{def}}{=} \mathcal{D}(T(\mathcal{I}(a),\mathcal{I}(b)))$.

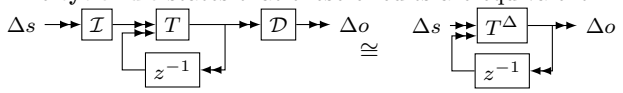
S^Δ has many nice properties:

The **chain rule** states that $(Q_1 \circ Q_2)^\Delta = {Q_1}^\Delta \circ {Q_2}^\Delta$, i.e., these circuits are equivalent:



In the database world, we can read this as: **to incrementalize a composite query you can incrementalize each sub-query independently**. This gives us a simple deterministic recipe reducing the incremental version of an arbitrary query to the incremental version of its primitive operators.

The **cycle rule** states that these circuits are equivalent:



(We have omitted the labels on the inputs of T.) In other words, the incremental version of a feedback loop around a query is just the feedback loop with the incremental query for its body. This result will be useful for recursive queries.

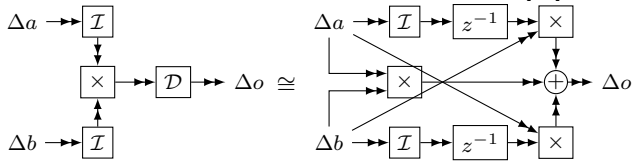
We call an operator S **linear** if it has the property that $S(a+b) = S(a)+S(b)$ (where $+$ is the addition of streams). For a linear operator S we have $S^\Delta = S$. This is very useful because many primitive database operations can be implemented as linear operators: selection, projection, filtering, grouping, parts of aggregation are all linear. Moreover, the following operators are linear: $-$, z^{-1}, $\mathcal{I}$, $\mathcal{D}$, $\uparrow f$ if f is a linear function.

We call an operator T with two inputs **bilinear** if it distributes over stream addition: $T(a+b,c) = T(a,c)+T(b,c)$, and $T(a,c+d) = T(a,c) + T(a,d)$. (Similar to multiplication's distributivity over addition.) In databases intersection, joins, and Cartesian products are bilinear.

Using infix notation, for a bilinear operator $\times$ we have:

$$(\Delta a \times \Delta b)^\Delta =$$
$$(\Delta a \times \Delta b \;+\; z^{-1}(\mathcal{I}(\Delta a)) \times \Delta b \;+\; \Delta a \times z^{-1}(\mathcal{I}(\Delta b)) =$$
$$\Delta a \times \Delta b + z^{-1}(a) \times \Delta b + \Delta a \times z^{-1}(b)$$

If we ignore the delay operators in this equation we recover the well-known formula for join delta queries, e.g.,[15].



What is the intuition behind this diagram? Let us consider the case of Cartesian product $a \times b$. The incremental product has inputs $\Delta a = \mathcal{D}(a)$ and $\Delta b = \mathcal{D}(b)$. What happens when we add a row x to relation a (i.e., $\Delta a = x$)? The new row x will appear in the output change combined with every row in the *previous version* of the *full* relation b. The operator $\mathcal{I}(\Delta b)$ in fact computes relation b from the stream Δb of changes, and z^{-1} applied to this value gives us its previous version. So the bottom $\times$ operator computes $x \times z^{-1}(b) = \Delta a \times z^{-1}(\mathcal{I}(\Delta b))$, the change produced by the new row x. The top $\times$ operator performs the symmetric operation for the changes of the b relation. The middle $\times$ operator produces the results of changes to both inputs.

4. IVM FOR THE RELATIONAL ALGEBRA

In this section we apply the results on incremental computation to relational databases. As explained in the introduction, our goal is to efficiently compute the incremental version of any relational query Q.

However, we face a technical problem: we said that streams require their values to belong to a commutative group, and relational databases in general are *not* commutative groups, since they operate on sets. Fortunately, there is a well-known tool in the database literature which converts set operations into group operations by using $\mathbb{Z}$-sets (also called z-relations [12]) to represent sets.

4.1 $\mathbb{Z}$-sets

$\mathbb{Z}$-sets generalize database tables: think of a $\mathbb{Z}$-set as a table where each row has an associated integer weight, possibly negative. This weight indicates *how many times* the row belongs to the table.

The following table shows an example $\mathbb{Z}$-set with three rows. The first row has value `joe` and weight 1. We do not show rows with weight 0.

Row	Weight
joe	1
mary	2
anne	-1

$\mathbb{Z}$-sets generalize sets and multisets: a set can be represented as a $\mathbb{Z}$-set by associating a weight of 1 with each element. Multisets (also called "bags" in the database literature) are $\mathbb{Z}$-sets where all weights are positive. Crucially, $\mathbb{Z}$-sets can also represent *changes* to sets and bags. Negative weights represent rows that are being *removed*.

We can define three operations on $\mathbb{Z}$-sets with values of a given type: (1) **zero** (a $\mathbb{Z}$-set with all weights 0) (2) **negation**: just negate all weights; (3) **plus**: add up the weights of the rows that have the same value. Using these operations $\mathbb{Z}$-sets are a commutative group.

We define the function *distinct* on $\mathbb{Z}$-sets. This function's output is a $\mathbb{Z}$-set where all rows of the input with negative weights are removed, and all positive weights are changed to 1. For example, the *distinct* of the above $\mathbb{Z}$-set is:

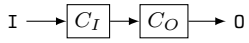
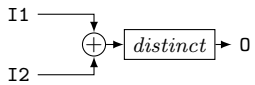
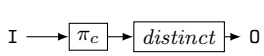
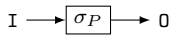
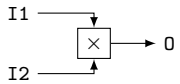
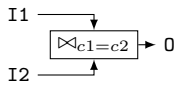
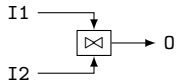
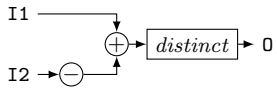
Row	Weight
joe	1
mary	1

Notice that *distinct* "removes" duplicates from multisets, and it also eliminates rows with negative weights.

4.2 Implementing relational operators

The fact that relational algebra can be implemented by computations on $\mathbb{Z}$-sets has been shown before, e.g. [13]. The translation of the relational operators to functions computing on $\mathbb{Z}$-sets is shown in Table 1. The functions (π, σ, $\bowtie$, $\times$) are the standard relational operators projection, selection, join, Cartesian product. The first row of the table

Table 1: Implementation of SQL relational set operators as circuits computing on $\mathbb{Z}$-sets.

Operation	SQL example	DBSP circuit	Details
Composition	`SELECT ... FROM (SELECT ... FROM I)`		C_I circuit for inner query, C_O circuit for outer query.
Union	`(SELECT * FROM I1) UNION (SELECT * FROM I2)`		*distinct* eliminates duplicates. An implementation of `UNION ALL` does not need the *distinct*.
Projection	`SELECT DISTINCT I.c FROM I`		Project each row with its weight unchanged. Add up weights of identical rows.
Filtering	`SELECT * FROM I WHERE P(...)`		P is a predicate applied to each row. Select each row separately. If the row is selected, preserve the weight, else make the weight 0.
Cartesian product	`SELECT I1.*, I2.* FROM I1, I2`		The weight of the pair (a,b) is the product of the the weights of a and b.
Equi-join	`SELECT I1.*, I2.* FROM I1 JOIN I2 ON I1.c1 = I2.c2`		Multiply the weights of the rows that appear in the output.
Intersection	`(SELECT * FROM I1) INTERSECT (SELECT * FROM I2)`		Special case of equi-join when both relations have the same schema.
Difference	`SELECT * FROM I1 EXCEPT SELECT * FROM I2`		*distinct* removes rows with negative weights from the result.

shows that a composite query is translated recursively: implement the sub-queries, and connect them with an arrow. This gives us a recipe for translating an arbitrary relational query plan into a circuit.

The translation is fairly straightforward, but many operators require the application of a *distinct* **to produce sets**. For example, $a \cup b = distinct(a + b)$, $a \setminus b = distinct(a - b)$. Filtering on $\mathbb{Z}$-sets works exactly as filtering on sets, but preserves the weight of each value. Selection on $\mathbb{Z}$-sets works similar to selection on sets, but also preserves the weights.

This is a faithful implementation of the relational algebra — the underlying mathematical theory that underlies modern databases — using $\mathbb{Z}$-sets. This implementation produces an abundance of *distinct* operators, but there are known optimizations for removing some of them.

The following functions in Table 1 are linear: $\sigma, \pi, -, +$. The following functions are bilinear: $\times, \bowtie$. In fact, the only non-linear function is *distinct*. In consequence, all these functions (lifted) have very efficient incremental versions.

To explain why these functions are linear, consider the filtering query from the introduction (`WHERE`). What is the change in the output when we add a new row to the input? It is sufficient to check the predicate for the new row. If the predicate returns `true`, the new row is added to the output. So the change in the output only depends on the change in the input, and not on the actual contents of the input. This is what makes the operation linear.

4.3 Incremental view maintenance

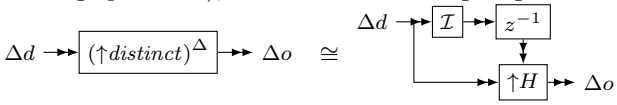
Let us consider a relational query Q defining a view V. To the following algorithm builds a DBSP circuit for Q^Δ:

ALGORITHM 4.1 (INCREMENTAL VIEW MAINTENANCE).

(1) *Translate Q into a circuit using the rules in Table 1.*
(2) *[Optional] Remove some distinct operations.*
(3) *Lift the whole circuit, converting it to a circuit operating on streams, using formula (**) in Section 2.*
(4) *Incrementalize the circuit "surrounding" it with $\mathcal{I}$ and $\mathcal{D}$.*
(5) *Apply the chain rule recursively, producing a circuit using only primitive incremental operations.*

This algorithm is deterministic; the running time is proportional to the number of operators in the query. Step (2) generates an equivalent circuit, with fewer *distinct* operators. Step (3) yields a circuit that consumes a *stream* of complete database snapshots and outputs a stream of view snapshots. Step (4) yields a circuit that consumes a stream of *database changes* and outputs a stream of *view changes*; however, the internal operation of the circuit is non-incremental, as it rebuilds the complete database using integrations. Step (5) optimizes the circuit by replacing each primitive operator with its incremental version. It essentially adds a $\mathcal{I} \circ \mathcal{D}$ pair on every edge in the circuit, and then uses the chain rule to replace each $\mathcal{I} \circ Q \circ \mathcal{D}$ with Q^Δ.

After running this algorithm, all primitives operations are replaced by their incremental versions. The only non-linear operation from Table 1 is *distinct*. However, there is an efficient incremental implementation for *distinct* (this construction has also been known before, but we show it in terms of streaming operations), shown in the following diagram:



The function H has two inputs: the left input is the change Δd, while the top input is the full set, obtained as an integral of the changes. H detects whether the weight of a row in the full set is changing sign (from negative to positive on a row insertion, and from positive to negative on a deletion) when the row appears in a new change. Here is the intuition why *distinct* is efficiently incrementalizable: only tuples that appear in the input change Δd can appear in the output change Δo, so the work performed is $O(|\Delta d|)$. The implementation needs to maintain the *entire input set* (similar to joins) in order to discover whether an item is new or not. That is the purpose of the $\mathcal{I}$ operator.

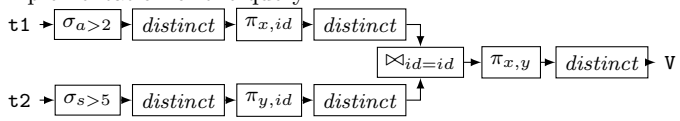
The algorithm reduces the problem of incremental execution of a query plan to the incremental execution of subplans/primitive operators. However, this algorithm works even if we use a primitive P for which no efficient incremental version is known: we can always use the inefficient "brute-force" implementation given by $P^{\Delta} = \mathcal{D} \circ \uparrow P \circ \mathcal{I}$.

4.4 Relational Query Example

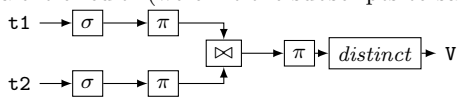
Let's apply the IVM algorithm to the following SQL query:

```
CREATE VIEW v AS
SELECT DISTINCT a.x, b.y FROM (
     SELECT t1.x, t1.id FROM t1 WHERE t1.a > 2
) a JOIN (
     SELECT t2.id, t2.y FROM t2 WHERE t2.s > 5
) b ON a.id = b.id
```

Step 1: Create a DBSP circuit to represent this query using the rules in Table 1; this circuit is essentially a dataflow implementation of the query:

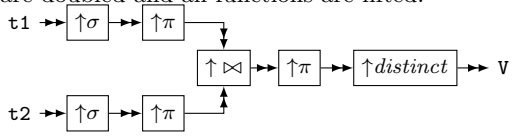


Step 2: eliminate *distinct* operators, producing an equivalent circuit: (we omit the subscripts to save space):

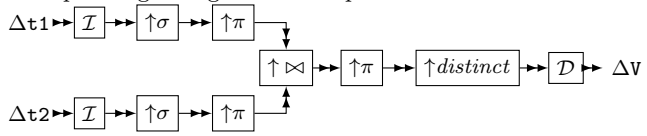


This step is used in some traditional database optimizers. Note that some arrows that represented sets in the original circuit may represent multisets in the optimized circuit.

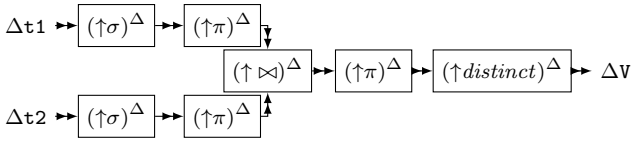
Step 3: lift the circuit to compute over streams; all arrows are doubled and all functions are lifted:



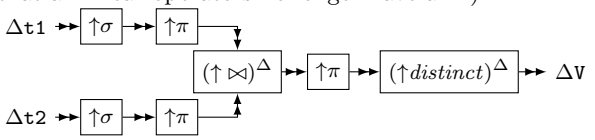
Step 4: incrementalize circuit, obtaining a circuit that computes over changes; this circuit receives changes to relations `t1` and `t2` and for each such change it produces the corresponding change in the output view `V`:



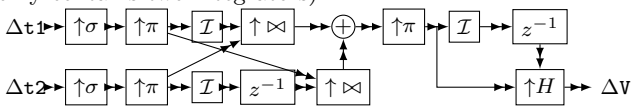
Step 5: apply the chain rule to rewrite the circuit as a composition of incremental operators; notice the use of $.^{\Delta}$ for all operators:



Use the linearity of σ and π to simplify this circuit (notice that all linear operators no longer have a $.^{\Delta}$):



Finally, replace the incremental join and the incremental *distinct*, with their incremental implementations, obtaining the following circuit (we have used a slightly different expansion for the join than the one shown previously; this one only contains two integrators):



Notice that the resulting circuit contains three integration operations: two from the join, and one from the *distinct*. It also contains two join operators. However, the work performed by each operator for each new input is proportional to the size of its input change.

4.5 SQL

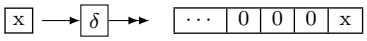
SQL is richer than the relational algebra. It can perform operations on multisets, and it offers operations such as `GROUP BY` and aggregations. All of these can be modeled as operations on $\mathbb{Z}$-set-like structures. Moreover, `GROUP BY` is a linear operation. Some aggregations are "almost" linear, but other, such as `MIN`, require maintaining the full input set, similar to *distinct*, to properly handle deletions. See the full paper and the technical report [6] for more details.

5. RECURSIVE QUERIES

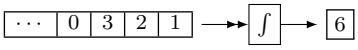
Recursive queries are very useful in many applications. For example, graph algorithms such as graph reachability or transitive closure are naturally expressed using recursive queries. We introduce two simple DBSP stream operators that are used for expressing recursive query evaluation. These operators allow us to build circuits implementing looping constructs, which are used to iterate computations until a fixed-point is reached (i.e., the output of some operator does not change anymore).

5.1 Creating and destroying streams

The **delta function** $\delta : A \to \mathcal{S}_A$ produces a stream from a scalar value. Given an input value x, the output stream is x followed by an infinite number of zeros. The input of δ has a single arrow, while the output has a double arrow.



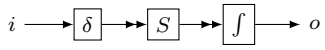
We define the function $\int : \mathcal{S}_A \to A$. Its input stream is required to eventually reach the value 0 and never change afterwards. This function just sums up all the values in the input stream and returns a single result when it encounters the first 0 in the input stream. Notice that the input is a double arrow, while the output is a single arrow. E.g.,:



(This function is also an integrator; its relationship to the $\mathcal{I}$ operator is the same one as the relationship of the definite integral [1] to the indefinite integral [2] in mathematics.)

δ and $\int$ are both linear.

So far we have used a tacit assumption that "time" is common for all streams in a program. For example, when we add two streams, we assume that they use the same "clock". However, the δ operator creates a stream with a "new", independent time dimension. We require *well-formed circuits* to "insulate" nested time domains by "bracketing" them between a δ and an $\int$ operator:



S is a streaming operator, but the entire circuit implements a scalar function, as shown by the single arrowheads.

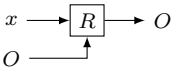
5.2 Implementing recursive queries

We describe the implementation of recursive queries in DBSP. SQL can only express very limited recursive queries, so here we model Datalog queries. In general, a Datalog program defines a set of mutually recursive relations.

We describe the algorithm to build DBSP circuits for the simple case of a single-input, single-output recursive query. The input of our algorithm is a Datalog query of the form $O = R(x, O)$, where R is a **relational, non-recursive** query, producing a set as a result, but whose output O is also an input. The output of the algorithm is a DBSP circuit which evaluates this recursive query producing output O when given the input x. In this section we build a non-incremental circuit, which evaluates the Datalog query; in Section 5.3 we derive the incremental version of this circuit.

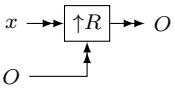
Algorithm 5.1 (recursive queries).

(1) *Implement the non-recursive relational query R as described in Section 4 and Table 1; this produces an acyclic circuit whose inputs and outputs are $\mathbb{Z}$-sets:*



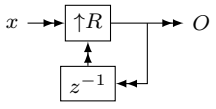
In all these diagrams we show input 0 of operator R on the left, and input 1 on the bottom.

(2) *Lift this circuit to operate on streams:*

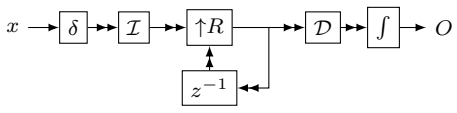


Construct $\uparrow R$ by lifting each operator individually, using equation (**) in Section 2.

(3) *Build a cycle, connecting the output to the corresponding recursive input via a delay:*



(4) *"Bracket" the circuit as follows:*



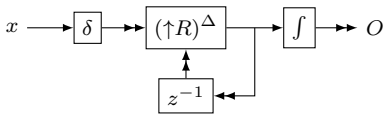
The left input of $\uparrow R$ is an infinite stream of identical values $\boxed{\cdots \mid x \mid x \mid x \mid x}$. The feedback cycle in this circuit is a `while` loop that iterates until no changes are observed (i.e., a fixed-point of R is reached); the outputs produced by $\uparrow R$ will be: in sequence $R(x, 0)$, $R(x, R(x, 0))$, $R(x, R(x, R(x, 0)))$, etc.. The $\mathcal{D}$ operator yields the set of *new changes* computed by each iteration of the loop. When the set of new changes becomes zero, the fixed point has been reached.

Please note that this is **not** a streaming circuit: the input and output arrows are both simple. This is a circuit which receives a single input value and produces a single corresponding output. The circuit uses streams internally to implement the fixed point iteration.

A concrete example for a transitive closure query is Section 8.2 of our technical report [6].

When R, the body of the loop, implements a Datalog programs computing on a finite data domain, this program can be proven to always terminate and compute the least fixed point that contains x. For an arbirary function R, the resulting circuit may loop forever for some inputs.

In fact, this circuit implements the standard Datalog **naïve evaluation** algorithm (e.g., see Algorithm 1 in [11]). Notice that the inner part of the circuit is the incremental form of another circuit, since it is sandwiched between $\mathcal{I}$ and $\mathcal{D}$ operators. Using the cycle rule we can rewrite this circuit:



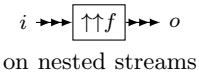
This circuit implements **semi-naïve evaluation** (Algorithm 2 in [11]). We have just proven the correctness of semi-naïve evaluation as an immediate consequence of the cycle rule!

5.3 Incremental recursive programs

In Section 2–4 we showed how to incrementalize a relational query by compiling it into a circuit, lifting the circuit to compute on streams, and applying the $\cdot^{\Delta}$ operator. In Section 5 we showed how to compile a recursive query into a circuit that employs incremental computation internally, to compute the fixed point. Here we combine these results to construct a circuit that evaluates a *recursive query incrementally*. The circuit receives a stream of updates to input relations, and for every update recomputes the fixed point. To do this incrementally, it preserves the stream of changes to recursive relations produced by the iterative fixed point computation, and adjusts this stream to account for the modified inputs. Thus, every element of the input stream yields a stream of adjustments to the fixed point computation, using *nested streams*.

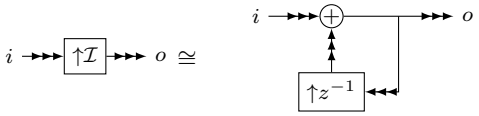
In the same way streams are infinite vectors, streams of streams are infinite matrices. We denote streams of streams with triple arrows in our diagrams.

The same way we lift functions to produce stream operators, we can lift stream operators to produce operators on streams of streams. A scalar function f can be lifted twice to produce an operator between streams of streams:

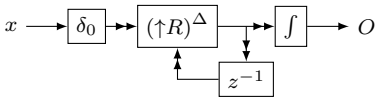


The operator z^{-1} on nested streams delays "rows" of the matrix, while $\uparrow z^{-1}$ delays "columns".

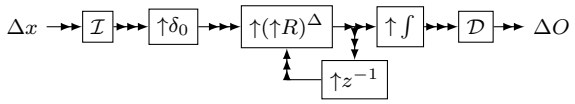
We have seen in equation (**) that lifting a graph entails lifting all operators. This extends to graphs with cycles, e.g:



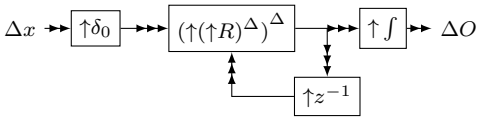
This gives us the ability to lift entire circuits, including circuits computing on streams and having feedback edges. With this machinery we can now apply Algorithm 4.1 to arbitrary circuits, even circuits for recursive relations.
Step 1: Start with the "semi-naive" circuit:



Step 2: nothing to do for *distinct*.
Steps 3 and 4: Lift the circuit and incrementalize:



Step 5: apply the chain rule and the linearity of $\uparrow\delta_0$ and $\uparrow\int$:



This is the incremental version of a recursive query. A concrete example for a transitive closure query is Section 9.1 of our technical report [6].

6. IMPLEMENTATION

DBSP does not make any simplifying assumptions that would make it impractical. In fact, Feldera Inc. has built an open-source implementation of DBSP as a query engine in Rust [9]; and also a compiler from SQL to DBSP [10]. This compiler handles essentially the entire SQL language. The compiler generates execution plans for incrementally maintaining any number of views defined in SQL.

Plan quality. A relational algebra query can be implemented by multiple plans, each with a different data-dependent cost. The input of Algorithm 4.1 is a non-incremental query plan, produced by a query planner. The algorithm produces an incremental plan that is "similar" to the input plan.

Standard query planners use cost-based heuristics and data statistics to optimize plans. A generic IVM planner does not have this luxury, since the plan must be generated *before* any data has been fed to the query. Nevertheless, all standard query optimization techniques, perhaps based on historical statistics, can be used to generate the query plan that is supplied to our Algorithm. The question of optimality in the context of IVM plan is a much more difficult topic than optimization of ad-hoc queries, since the chosen IVM plan will execute for *all future database updates*.

Tradeoffs. Incremental computation is not free. It is in fact a trade-off between time and space. In the cost analysis we have to consider both the time and the space used by each operator. While many incremental database operations can be implemented using work proportional to the size of the changes, and no storage overhead, several classes of database operations, such as joins, "distinct", and aggregates can be implemented efficiently only using additional storage in the form of *indexes*. The size of these indexes is proportional to the size of the total data in the database (and not just to the size of the changes) — and since some indexes are over intermediate relations, they can even exceed the size of the original database. In DBSP the indexes are represented by delay operators z^{-1}. In fact, the delay operator (and its lifted variant $\uparrow z^{-1}$) are the only operators that maintains state. This is also the only state that needs to be persisted, checkpointed, or migrated to make DBSP computations fault-tolerant.

DBSP is an "eager" or "top-down" execution model: it constantly maintains the entire contents of any number of views, even if no one really wants to inspect the views. In contrast, "lazy" or "bottom-up" models only build part of the views when the views are inspected. Such models have the potential to be more efficient. Eager models can be converted into lazy ones if something is known about the class of operations that will be executed against the views.

Start-up costs. When a new view is installed, the IVM system must compute the first change, which is the same as the initial contents of the view. This computation is in proportional to the size of the whole database. This is known as the "backfill" problem. Likewise, changes to the definition of a view or the data schema require recomputing the affected queries from scratch.

Adopting DBSP. Traditional databases do not offer efficient IVM implementations for arbitrary queries. Databases could in principle be retrofitted to use the algorithms in this paper, but the existing query engines are not built around structures that can represent negative changes (like $\mathbb{Z}$-sets), so this effort will require a significant redesign.

Moreover, we argue that databases should not only compute views incrementally, but should use "changes" as the fundamental data structure to communicate with their environment: a database service should offer the following API: users register to receive notifications for changes in one or more views. Then, for any transaction committed, each user receives a notification containing the list of changes for the all the views they registered. Databases today do not have convenient mechanism for reporting changes to the outside world. In fact, entire industries have sprung up around the concept of Change Data Capture [3], which is building ad-hoc solutions for extracting changes from databases, usually by inspecting the write-ahead transaction log.

7. CONCLUSIONS

We have introduced DBSP, a model of computation based on infinite streams over commutative groups. In this model streams are used for 3 purposes: (1) to model consecutive snapshots of a database, (2) to model consecutive changes (deltas, or transactions) applied to a database and changes of a maintained view, (3) to model consecutive values of loop-carried variables in recursive computations.

We have defined an abstract notion of incremental computation over streams, and defined the incrementalization operator $\cdot^\Delta$, which transforms an *arbitrary* stream computation Q into its incremental version Q^Δ. The incrementalization operator has some very nice algebraic properties, which gave us a general algorithm for incrementalizing many classes of complex queries, including arbitrary recursive queries.

We believe that DBSP can form a solid foundation for a theory and practice of streaming incremental computation.

8. REFERENCES

[1] `https://en.wikipedia.org/wiki/Integral`. Retrieved March 2024.

[2] `https://en.wikipedia.org/wiki/Antiderivative`. Retrieved March 2024.

[3] `https://en.wikipedia.orag/wiki/Change_data_capture`. Retrieved March 2024.

[4] Causal system. `https://en.wikipedia.org/wiki/Causal_system`. Retrieved March 2024.

[5] M. Budiu, T. Chajed, F. McSherry, L. Ryzhyk, and V. Tannen. DBSP: Automatic incremental view maintenance for rich query languages. In *Proceedings of the VLDB Endowment (VLDB)*, volume 16, pages 1601–1614, Vancouver, Canada, August 2023. Best paper award.

[6] M. Budiu, F. McSherry, L. Ryzhyk, and V. Tannen. DBSP: A language for expressing incremental view maintenance for rich query languages. https://github.com/feldera/feldera/blob/main/papers/spec.pdf, December 2022.

[7] T. Chajed. DBSP formalization. `https://github.com/tchajed/dbsp-theory`, Dec. 2022.

[8] L. de Moura, S. Kong, J. Avigad, F. van Doorn, and J. von Raumer. The Lean theorem prover. In *International Conference on Automated Deduction (CADE-25)*, Berlin, Germany, 2015.

[9] Feldera Inc. DBSP Rust crate. `https://crates.com/crates/dbsp`. Retrieved March 2024.

[10] Feldera Inc. SQL to DBSP compiler. `https://github.com/feldera/feldera/tree/main/sql-to-dbsp-compiler`. Retrieved March 2024.

[11] S. Greco and C. Molinaro. Datalog and logic databases. *Synthesis Lectures on Data Management*, 7(2):1–169, 2015.

[12] T. J. Green, Z. G. Ives, and V. Tannen. Reconcilable differences. *Theory of Computing Systems*, 49(2):460–488, 2011.

[13] T. J. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *Symposium on Principles of Database Systems (PODS)*, page 31–40, Beijing, China, June 11-14 2007.

[14] A. Gupta, I. S. Mumick, et al. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Eng. Bull.*, 18(2):3–18, 1995.

[15] C. Koch. Incremental query evaluation in a ring of databases. In *Symposium on Principles of Database Systems (PODS)*, page 87–98, Indianapolis, Indiana, USA, 2010.

[16] L. R. Rabiner and B. Gold, editors. *Theory and Application of Digital Signal Processing*. Prentice-Hall, 1975.