SIGMOD Officers, Committees, and Awardees

Chair
Juliana Freire
Computer Science & Engineering
New York University
Brooklyn, New York
USA
+1 646 997 4128
juliana.freire <at> nyu.edu

Vice-Chair
Ihab Francis Ilyas
Cheriton School of Computer Science
University of Waterloo
Waterloo, Ontario
CANADA
+1 519 888 4567 ext. 33145
ilyas <at> uwaterloo.ca

Secretary/Treasurer
Fatma Ozcan
Google
Sunnyvale
California
USA
fozcan <at> google.com

SIGMOD Executive Committee:
Juliana Freire (Chair), Ihab Francis Ilyas (Vice-Chair), Fatma Ozcan (Treasurer), K. Selçuk Candan, Rada Chirkova, Chris Jermaine, Wang-Chiew Tan, AnHai Doan, Leonid Libkin, and Curtis Dyreson

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 Director:
Curtis Dyreson, Utah State University

Associate Information Directors:
Huiping Cao, Georgia Koutrika, Wim Martens, and Sourav S Bhowmick

SIGMOD Record Editor-in-Chief:
Rada Chirkova, NC State University

SIGMOD Record Associate Editors:
Azza Abouzied, Lyublena Antova, Marcelo Arenas, Renata Borovica-Gajic, Vanessa Braganholo, Aaron J. Elmore, Wim Martens, Kyriakos Mouratidis, Dan Olteanu, Divesh Srivastava, Immanuel Trummer, Yannis Velegrakis, Marianne Winslett, and Jun Yang

SIGMOD Conference Coordinator:
K. Selçuk Candan, Arizona State University

PODS Executive Committee:
Dan Suciu (Chair), Tova Milo, Diego Calvanese, Wang-Chiew Tan, Rick Hull, and Floris Geerts

Sister Society Liaisons:
Raghu Ramakhrishnan (SIGKDD), Yannis Ioannidis (EDBT Endowment), Christian Jensen (IEEE TKDE)

SIGMOD Awards Committee:
M. Tamer Özsu (Chair), Stefano Ceri, Yanlei Diao, Volker Markl, Renee Miller, and Sunita Sarawagi

Jim Gray Doctoral Dissertation Award Committee:
Pınar Tözün (co-Chair), Viktor Leis (co-Chair), Peter Bailis, Alexandra Meliou, Bailu Ding, Vanessa Braganholo, Immanuel Trummer, and Joy Arulraj

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)
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)
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)

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)

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)
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)
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)

SIGMOD Jim Gray Doctoral Dissertation Award

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

- **2006** *Winner*: Gerome Miklau. *Honorable Mentions*: Marcelo Arenas and Yanlei Diao
- **2007** *Winner*: Boon Thau Loo. *Honorable Mentions*: Xifeng Yan and Martin Theobald
- **2008** *Winner*: Ariel Fuxman. *Honorable Mentions*: Cong Yu and Nilesh Dalvi
- **2009** *Winner*: Daniel Abadi. *Honorable Mentions*: Bee-Chung Chen and Ashwin Machanavajjhala
- **2010** *Winner:* Christopher Ré. *Honorable Mentions*: Soumyadeb Mitra and Fabian Suchanek
- **2011** *Winner*: Stratos Idreos. *Honorable Mentions*: Todd Green and Karl Schnaitterz

- **2012** *Winner*: Ryan Johnson. *Honorable Mention*: Bogdan Alexe
- **2013** *Winner*: Sudipto Das, *Honorable Mention*: Herodotos Herodotou and Wenchao Zhou
- **2014** *Winners*: Aditya Parameswaran and Andy Pavlo.
- **2015** *Winner*: Alexander Thomson. *Honorable Mentions*: Marina Drosou and Karthik Ramachandra
- **2016** *Winner*: Paris Koutris. *Honorable Mentions*: Pinar Tozun and Alvin Cheung
- **2017** *Winner*: Peter Bailis. *Honorable Mention*: Immanuel Trummer
- **2018** *Winner*: Viktor Leis. *Honorable Mention*: Luis Galárraga and Yongjoo Park
- **2019** *Winner*: Joy Arulraj. *Honorable Mention*: Bas Ketsman
- **2020** *Winner:* Jose Faleiro. *Honorable Mention:* Silu Huang

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

[Last updated: March 31, 2021]

Guest Editor's Notes

Welcome to the March 2021 issue of the ACM SIGMOD Record!

The new year of 2021 begins with a special issue on the **2020 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, 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 Marcelo Arenas, Rada Chirkova, Wim Martens, Jun Yang, and Divesh Srivastava. We solicited articles from PODS 2020, SIGMOD 2020, VLDB 2020, ICDE 2020, EDBT 2020, and ICDT 2020, as well as from community nominations. Through a careful review process ten articles were finally selected as 2020 Research Highlights. 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 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 2020 research highlights cover a broad set of topics, including (a) a substantial advance on the question of reliability of streaming algorithms when the input to the algorithm might depend on the algorithm's earlier output ("A Framework for Adversarially Robust Streaming Algorithms"); (b) key innovations in processing database transactions driven by new high-bandwidth communication technologies that are becoming common in data centers ("Chiller: Contention-centric Transaction Execution and Data Partitioning for Modern Networks"); (c) a versatile framework for automatic extraction of benchmarks and their distributed execution and performance monitoring ("DIAMetrics: Benchmarking Query Engines at Scale"); (d) non-trivial results on finding a 2-approximate solution to the problem of directed densest subgraph discovery using a new concept of [x, y]-core ("Efficient Directed Densest Subgraph Discovery"); (e) insightful work on data structures for similarity search that support algorithmic fairness ("Fair Near Neighbor Search via Sampling"); (f) a general abstraction, along with advanced interfaces, focusing on effective visual search ("From Sketching to Natural Language: Expressive Visual Querying for Accelerating Insight"); (g) practical improvements in hash tables for string processing in real system implementations ("Optimistically Compressed Hash Tables & Strings in the USSR"); (h) answers to foundational questions of the representation and querying of infinite probabilistic databases ("Probabilistic Data with Continuous Distributions"); (i) an elegant quantitative approach to defining and computing explanation scores, based on the Shapley value of cooperative games ("Query Games in Databases"); and (j) design and evaluation of a persistent dynamic hash table using Optane byte-addressable non-volatile memory ("Scaling Dynamic Hash Tables on Real Persistent Memory").

On behalf of the SIGMOD Record Editorial Board, I hope that you enjoy reading the March 2021 issue of the SIGMOD Record!

Divesh Srivastava

March 2021

Technical Perspective: A Framework for Adversarially Robust Streaming Algorithms

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

Over the past two decades the data management community has devoted particular attention to handling data that arrives as a stream of updates. This captures a number of "big data" scenarios, ranging from monitoring networks to processing high volumes of transactions in commerce and finance. This has led to data streams becoming a mainstream data management topic, with many systems offering explicit support for handling such inputs. Within these systems, streaming algorithms are used to approximate various statistical and modeling queries, which would traditionally require random access to the full data to compute exactly.

The area of streaming algorithms is by now very mature, with effective and efficient techniques known for many core analytics problems – tracking frequencies of fluctuating quantities, monitoring statistical distributions – supporting applications like anomaly and change detection over high speed, multidimensional streams. These algorithms have been deployed deep within the infrastructure of major data processors – governments, search engines and cloud service providers – to flag anomalies and changes in distribution. These applications consequently demand a high level of reliability to ensure that the results can be trusted.

High reliability comes assured when an algorithm provides a deterministic guarantee. However, many of the widely used streaming algorithms are randomized in nature: for any input, there is some small probability that they will give an incorrect answer. This prompts the question of how reliable can these algorithms be when they are queried multiple times in succession? In particular, what happens if the input to the algorithm might be set by actions taken in response to the algorithm's earlier output – then the prior fixed input guarantees may not hold.

The following paper, "A Framework for Adversarially Robust Streaming Algorithms", makes a substantial advance on this question by addressing the notion of "adversarially robust" streaming algorithms. The idea is to show a strong guarantee on a streaming algorithm by assuming that the input is generated by an adaptive adversary, who observes the outputs of the algorithm, and sends new stream elements with the intent of provoking an erroneous response. If we can prove an algorithm is robust to such a malicious adversary, then we can be assured that it will be highly reliable on any input distribution, no matter how it is formed. A significant feature is that rather than consider the robustness of individual algorithms, the paper provides a framework so that *any* streaming algorithm for any problem can be made robust, by executing it with modified parameters or by running multiple independent instances. This is a powerful first step in measuring the "price of robustness", compared to solutions for the same problems which lack a robustness guarantee.

The topic of robust streaming has led to a flurry of activity in a short time. Recent interest in robust streaming stems in part from work of a subset of the paper's authors, who addressed the question of sampling, and showed that, after appropriate setting of parameters, several sampling algorithms are inherently robust [2]. Due to a temporal quirk, the results on sampling and streaming both appeared in PODS 2020, where the streaming work won the best paper award.

Subsequent work has sought to better understand the generality of robustness results, and to reduce the space needed to give these guarantees. New papers have sketched out an intriguing connection with differential privacy, to show that adding suitable noise to the output of an algorithm can ensure robustness [3]; they have provided tighter bounds for specific questions in the infinite window and sliding window model of streams [5]; and, most recently, proved a separation between the adversarial and non-adversarial model by showing that the "adaptive data analysis" problem which requires exponentially more space in the adversarial setting [4]; and exhibited a deep connection between adversarial sampling and the theory of online learning [1]. These rapid advances demonstrate the interest in this work, and suggest that there is still yet more to discover about this challenging topic.

1. REFERENCES

[1] N. Alon, O. Ben-Eliezer, Y. Dagan, S. Moran, M. Naor, and E. Yogev. Adversarial laws of large numbers and optimal regret in online classification. *CoRR*, abs/2101.09054, 2021. To appear in STOC 2021.

[2] O. Ben-Eliezer and E. Yogev. The adversarial robustness of sampling. In *ACM PODS*, 2020.

[3] A. Hassidim, H. Kaplan, Y. Mansour, Y. Matias, and U. Stemmer. Adversarially robust streaming algorithms via differential privacy. In *NeurIPS*, 2020.

[4] H. Kaplan, Y. Mansour, K. Nissim, and U. Stemmer. Separating adaptive streaming from oblivious streaming. *CoRR*, abs/2101.10836, 2021.

[5] D. P. Woodruff and S. Zhou. Tight bounds for adversarially robust streams and sliding windows via difference estimators. *CoRR*, abs/2011.07471, 2020.

A Framework for Adversarially Robust Streaming Algorithms

[Extended Abstract]

Omri Ben-Eliezer
Harvard University
omribene@cmsa.fas.harvard.edu

Rajesh Jayaram
Carnegie Mellon University
rkjayara@cs.cmu.edu

David P. Woodruff
Carnegie Mellon University
dwoodruf@cs.cmu.edu

Eylon Yogev
Boston Univ. & Tel Aviv Univ.
eylony@gmail.com

ABSTRACT

We investigate the adversarial robustness of streaming algorithms. In this context, an algorithm is considered robust if its performance guarantees hold even if the stream is chosen adaptively by an adversary that observes the outputs of the algorithm along the stream and can react in an online manner. While deterministic streaming algorithms are inherently robust, many central problems in the streaming literature do not admit sublinear-space deterministic algorithms; on the other hand, classical space-efficient randomized algorithms for these problems are generally not adversarially robust. This raises the natural question of whether there exist efficient adversarially robust (randomized) streaming algorithms for these problems.

In this work, we show that the answer is positive for various important streaming problems in the insertion-only model, including distinct elements and more generally F_p-estimation, F_p-heavy hitters, entropy estimation, and others. For all of these problems, we develop adversarially robust $(1+\varepsilon)$-approximation algorithms whose required space matches that of the best known non-robust algorithms up to a $\mathrm{poly}(\log n, 1/\varepsilon)$ multiplicative factor (and in some cases even up to a constant factor). Towards this end, we develop several generic tools allowing one to efficiently transform a non-robust streaming algorithm into a robust one in various scenarios.

1. INTRODUCTION

The streaming model of computation is a central and crucial tool for the analysis of massive datasets, where the sheer size of the input imposes stringent restrictions on the memory, computation time, and other resources available to the algorithms. Examples of practical settings where streaming algorithms are in need are easy to encounter. These include internet routers and traffic logs, databases, sensor networks, financial transaction data, and scientific data streams. Given this wide range of applicability, there has been significant effort devoted to designing and analyzing extremely efficient one-pass algorithms. We recommend the survey of [26] for a comprehensive overview of streaming algorithms and their applications.

Many central problems in the streaming literature do not admit sublinear-space deterministic algorithms, and in these cases randomized solutions are necessary. In other cases, randomized solutions are more efficient and simpler to implement than their deterministic counterparts. While randomized streaming algorithms are well-studied, the vast majority of them are defined and analyzed in the *static* setting, where the stream is worst-case but fixed in advance, and only then the randomness of the algorithm is chosen. However, assuming that the stream sequence is independent of the chosen randomness, and in particular that future elements of the stream do not depend on previous outputs of the streaming algorithm, may not be realistic [1, 4, 13, 14, 16, 25, 27], even in non-adversarial settings. For example, suppose that a user sequentially makes queries to a database, and receives an immediate response after each query. Naturally, future queries made by the user in such a setting may heavily depend on the responses given by the database to previous queries. In other words, the stream updates are chosen adaptively, and cannot be assumed to be fixed in advance.

A streaming algorithm that works even when the stream is adaptively chosen by an adversary (the precise definition given next) is said to be *adversarially robust*. Deterministic algorithms are inherently adversarially robust, since they are guaranteed to be correct on all possible inputs. However, the large gap in performance between deterministic and randomized streaming algorithms for many problems motivates the need for designing adversarially robust randomized algorithms, if they even exist. In particular, we would like to design adversarially robust randomized algorithms which are as space and time efficient as their static counterparts, and yet as robust as deterministic algorithms. The study of

This is a minor revision of the paper entitled "A Framework for Adversarially Robust Streaming Algorithms", published in the Proceedings of the 39th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS'20), June 14–19, 2020, Portland, OR, USA. ACM ISBN 978-1-4503-7108-7/20/06.
http://dx.doi.org/10.1145/3375395.3387658

such algorithms is the main focus of our work.

The Adversarial Setting.

There are several ways to define the adversarial setting, which may depend on the information the adversary (who chooses the stream) can observe from the streaming algorithm, as well as other restrictions imposed on the adversary. For the most part, we consider a general model, where the adversary is allowed unbounded computational power and resources, though we do discuss the case later when the adversary is computationally bounded. At each point in time, the streaming algorithm publishes its output to a query for the stream. The adversary observes these outputs one-by-one, and can choose the next update to the stream adaptively, depending on the full history of the outputs and stream updates. The goal of the adversary is to force the streaming algorithm to eventually produce an *incorrect* output to the query, as defined by the specific streaming problem in question.

Formally, a data stream of length m over a domain $[n]$ is a sequence of updates of the form $(a_1, \Delta_1), \ldots, (a_m, \Delta_m)$ where $a_t \in [n]$ is an index and $\Delta_t \in \mathbb{Z}$ is an increment or decrement to that index. The *frequency vector* $f \in \mathbb{R}^n$ of the stream is the vector with i^{th} coordinate $f_i = \sum_{t:a_t=i} \Delta_t$. We write $f^{(t)}$ to denote the frequency vector restricted to the first t updates, namely $f_i^{(t)} = \sum_{j \le t:a_j=i} \Delta_j$. It is assumed at all points t that the maximum coordinate in absolute value, denoted $\|f^{(t)}\|_\infty$, is at most M for some $M > 0$, and that $\log(mM) = O(\log n)$. In the *insertion-only* model, the updates are assumed to be positive, meaning $\Delta_t > 0$, whereas in the *turnstile* model Δ_t can be positive or negative.

The general task in streaming is to respond to some query $\mathcal{Q}$ about the frequency vector $f^{(t)}$ at each point in time $t \in [m]$. Oftentimes, this query is to approximate some function $g : \mathbb{R}^n \to \mathbb{R}$ of $f^{(t)}$ (ideally, one might wish to exactly compute the function g; however, in many cases, and in particular for the problems that we consider here, exact computation cannot be done with sublinear space). For example, counting the number of distinct elements in a data stream is among the most fundamental problems in the streaming literature; here $g(f^{(t)})$ is the number of non-zero entries in $f^{(t)}$. Since exact computation cannot be done in sublinear space [8], the goal is to approximate the value of $g(f^{(t)})$ to within a multiplicative factor of $(1 \pm \varepsilon)$. Another important streaming problem (which is not directly an estimation task) is the *Heavy-Hitters* problem, where the algorithm is tasked with finding all the coordinates in $f^{(t)}$ which are larger than some threshold τ.

Formally, the adversarial setting is modeled by a two-player game between a (randomized) StreamingAlgorithm and an Adversary. At the beginning, a query $\mathcal{Q}$ is fixed, which the StreamingAlgorithm must continually reply to. The game proceeds in rounds, where in the t-th round:

1. Adversary chooses an update $u_t = (a_t, \Delta_t)$ for the stream, which can depend, in particular, on all previous stream updates and outputs of StreamingAlgorithm.
2. StreamingAlgorithm processes the new update u_t and outputs its current response R^t to the query $\mathcal{Q}$.
3. Adversary observes R^t (stores it) and proceeds to the next round.

The goal of the Adversary is to make the StreamingAlgorithm output an incorrect response R^t to $\mathcal{Q}$ at some point t in the stream. For example, in the distinct elements problem, the adversary's goal is that on some step t, the estimate R^t will fail to be a $(1+\varepsilon)$-approximation of the true current number of distinct elements $|\{i \in [n] : f_i^{(t)} \neq 0\}|$.

Streaming algorithms in the adversarial setting.

It was shown by Hardt and Woodruff [16] that linear sketches are inherently *non-robust* in adversarial settings for a large family of problems, thus demonstrating a major limitation of such sketches. In particular, their results imply that no linear sketch can approximate the Euclidean norm of its input to within a polynomial multiplicative factor in the adversarial (turnstile) setting. Here, a linear sketch is an algorithm whose output depends only on values $S \cdot f$ and S, for some (usually randomized) sketching matrix $S \in \mathbb{R}^{k \times n}$. This is quite unfortunate, as the vast majority of turnstile streaming algorithms are in fact linear sketches.

Indeed, the typical guarantee is that for any fixed f, $S \cdot f$ satisfies some property with good probability. If f is allowed to depend on S, this property typically does not hold. For example, if S is a random Gaussian matrix, then the Euclidean norm $\|S \cdot f\|_2$ is close to $\|f\|_2$ with large probability. However, if f is allowed to depend on S, then one can choose f to be a large non-zero vector orthogonal to the rows of S, so that $\|S \cdot f\|_2$ is zero while $\|f\|_2$ is non-zero. One can show that for a number of sketches, answers to previous queries reveal information about S, and consequently an adversary can later construct an f, depending on S, to break them.

On the positive side, recent works of Ben-Eliezer and Yogev [4] and Alon et al. [1] show that *random sampling* is quite robust in the adaptive adversarial setting, albeit with a slightly larger sample size. While uniform sampling is a rather generic and useful tool, it is not sufficient for solving many important streaming tasks, such as estimating frequency moments (F_p-estimation), finding L_2 heavy hitters, and various other data analysis problems. This raises the natural question of whether there exist efficient adversarially robust randomized streaming algorithms for these problems and others, which is the main focus of this work. Perhaps even more importantly, we ask the following.

Is there a generic technique to transform a static streaming algorithm into an adversarially robust streaming algorithm?

This work answers the above questions affirmatively for a large class of algorithms.

1.1 Our Results

We devise adversarially robust algorithms for various fundamental insertion-only streaming problems, including distinct element estimation, F_p moment estimation, heavy hitters, entropy estimation, and several others. In addition, we give adversarially robust streaming algorithms which can handle a bounded number of deletions as well. The required space of our adversarially robust algorithms matches that of the best known non-robust ones up to a small multiplicative factor. Our new algorithmic results are summarized in Table 1.

In contrast, we demonstrate that some classical randomized algorithms for streaming problems in the static setting,

Table 1: A summary of our adversarially robust algorithms (in bold), as compared to the best known upper bounds for randomized algorithms in the static setting and lower bounds for deterministic algorithms. Note that all stated algorithms provide tracking. All results except for the last two (which hold in restricted versions of the turnstile model) are for insertion only streams. We write $\tilde{O}, \tilde{\Omega}$ to hide $\log \varepsilon^{-1}$ and $\log \log n$ factors. The lower bound for deterministic entropy estimation follows from a reduction from estimating F_p for $p = 1 + \tilde{\Theta}(\frac{\varepsilon}{\log^2 n})$.

Problem	Static Randomized	Deterministic	Adversarial	Comments
Distinct elements (F_0-estimation)	$\tilde{O}(\varepsilon^{-2} + \log n)$ [5]	$\Omega(n)$ [8]	$\boldsymbol{\tilde{O}(\varepsilon^{-3} + \varepsilon^{-1} \log n)}$	
			$\boldsymbol{\tilde{O}(\varepsilon^{-2} + \log n)}$	crypto/random oracle
F_p-estimation, $p \in (0,2] \setminus \{1\}$	$O(\varepsilon^{-2} \log n)$ [6]	$\tilde{\Omega}(2^{-1/(1-p)} \cdot n)$ [8]	$\boldsymbol{\tilde{O}(\varepsilon^{-3} \log n)}$	
	$O(\varepsilon^{-3} \log^2 n)$ [22]		$\boldsymbol{\tilde{O}(\varepsilon^{-3} \log^3 n)}$	$\delta = \Theta(n^{-(1/\varepsilon)\log n})$
F_p-estimation, $p > 2$	$O\Big(n^{1-\frac{2}{p}}(\varepsilon^{-3}\log^2 n + \varepsilon^{-\frac{6}{p}} \log^{\frac{4}{p}+1} n)\Big)$ [12]	$\Omega(n)$ [8]	$\boldsymbol{O\Big(n^{1-\frac{2}{p}}(\varepsilon^{-3}\log^2 n + \varepsilon^{-\frac{6}{p}} \log^{\frac{4}{p}+1} n)\Big)}$	$\delta = \Theta(n^{-(1/\varepsilon)\log n})$
ℓ_2 Heavy Hitters	$O(\varepsilon^{-2} \log^2 n)$ [7]	$\Omega(\sqrt{n})$ [21]	$\boldsymbol{\tilde{O}(\varepsilon^{-3} \log^2 n)}$	
Entropy Estimation	$O(\varepsilon^{-2} \log^3 n)$ [9]	$\tilde{\Omega}(n)$ [17]	$\boldsymbol{O(\varepsilon^{-5} \log^6 n)}$	
	$\tilde{O}(\varepsilon^{-2})$ [20]		$\boldsymbol{O(\varepsilon^{-5} \log^4 n)}$	crypto/random oracle
Turnstile F_p-estimation, $p \in (0,2]$	$O(\varepsilon^{-2} \lambda \log^2 n)$ [22]	$\Omega(n)$ [2]	$\boldsymbol{O(\varepsilon^{-2} \lambda \log^2 n)}$	λ-bounded F_p flip number, $\delta = \Theta(n^{-\lambda})$
F_p-estimation, $p \in [1,2]$, α-bounded deletions	$\tilde{O}(\varepsilon^{-2} \log \alpha \log n + \log^2 n)$ [19]	$\tilde{\Omega}(2^{-1/(1-p)} \cdot n)$ [8]	$\boldsymbol{O(\alpha \varepsilon^{-(2+p)} \log^3 n)}$	static only for $p = 1$

such as the celebrated Alon-Matias-Szegedy (AMS) sketch [2] for F_2-estimation, are inherently non-robust to adaptive adversarial attacks in a strong sense.

The Robustification Framework: Flip number, Sketch Switching, and Computation Paths.

Our adversarially robust algorithms make use of two generic robustification frameworks that we develop, allowing one to efficiently transform a non-robust streaming algorithm into a robust one in various settings. Both of the robustification methods rely on the fact that functions of interest do not drastically change their value too many times along the stream. Specifically, the transformed algorithms have space dependency on the *flip-number* of the stream, which is a bound on the number of times the function $g(f^{(t)})$ can change by a factor of $(1 \pm \varepsilon)$ in the stream (see Section 2).

The first method, called *sketch switching*, maintains multiple instances of the non-robust algorithm and switches between them in a way that cannot be exploited by the adversary. The second technique bounds the number of *computation paths* possible in the two-player adversarial game. This technique maintains only one copy of a non-robust algorithm, albeit with an extremely small probability of error δ. We show that a carefully rounded sequence of outputs generates only a small number of possible computation paths, which can then be used to ensure robustness by union bounding over these paths. The framework is described in Section 2.

The two above methods are incomparable: for some streaming problems the former is more efficient, while for others, the latter performs better, and we show examples of each. Specifically, sketch switching can exploit efficiency gains of *strong-tracking*, resulting in particularly good performance for static algorithms that can respond correctly to queries at each step without having to union bound over all m steps. In contrast, the computation paths technique can exploit an algorithm with good dependency on δ (the failure probability). Namely, algorithms that have small dependency in update-time or space on δ will benefit from the computation paths technique.

For each of the problems we consider, we show how to use the framework, in addition to some further techniques which we develop along the way, to solve it. Interestingly, we also demonstrate how cryptographic assumptions (which were not commonly used before in the streaming context) can be applied to obtain an adversarially robust algorithm against computationally bounded adversaries for the distinct elements problem at essentially no extra cost over the space optimal non-robust one. See Table 1 for a summary of our results in the adversarial setting compared to the state-of-the-art in the static setting, as well as to deterministic algorithms.

Distinct elements, F_p-estimation, and more.

Our first suite of results provides robust streaming algorithms for estimating F_p, the p^{th} frequency moment of the frequency vector, defined as $F_p = \|f\|_p^p = \sum_{i=1}^n |f_i|^p$, where we interpret $0^0 = 0$. Estimating frequency moments has a myriad of applications in databases, computer networks, data mining, and other contexts. Efficient algorithms for estimating distinct elements (i.e., estimating F_0) are important for databases, since query optimizers can use them to find the number of unique values of an attribute without having to perform an expensive sort on the values. Efficient algorithms for F_2 are useful for determining the output size of self-joins in databases, and for computing the surprise index of a data sequence [15]. Higher frequency moments are used to determine data skewness, which is important in parallel database applications [10].

We remark that for any fixed $p \neq 1$, including $p = 0$, any deterministic insertion-only algorithm for F_p-estimation requires $\Omega(n)$ space [2, 8] (note that for the case $p = 1$ there is a trivial $O(\log n)$-bit insertion only F_p-estimation algorithm: keeping a counter for $\sum_t \Delta_t$). In contrast, we show that randomized adversarially robust algorithms exist for all p, whose space complexity either matches or has a small

multiplicative overhead over the best static randomized algorithms.

We utilize an optimized version of the sketch switching method to derive an upper bound for estimating the number of distinct elements. The result is an adversarially robust F_0 estimation algorithm whose complexity is only a $\Theta(\frac{1}{\varepsilon}\log\varepsilon^{-1})$ factor larger than the optimal static (non-robust) algorithm [5].

THEOREM 1.1. *There is an algorithm which, when run on an adversarial insertion only stream, with probability at least $1-\delta$ produces at every step $t \in [m]$ an estimate R^t such that $R^t = (1\pm\varepsilon)\|f^{(t)}\|_0$. The space used by the algorithm is*

$$O\left(\frac{\log(1/\varepsilon)}{\varepsilon}\left(\frac{\log\varepsilon^{-1}+\log\delta^{-1}+\log\log n}{\varepsilon^2}+\log n\right)\right).$$

A second result applies the computation paths method with a new *static* algorithm for F_0 estimation which has very small update-time dependency on δ, and nearly optimal space complexity. As a result, we obtain an adversarially robust F_0 estimation algorithm with extremely fast update time (note that the update time of the above sketch switching algorithm would be $O(\varepsilon^{-1}\log n)$ to obtain the same result, even for constant δ).

A third result takes a different approach: it shows that under certain standard cryptographic assumptions, there exists an adversarially robust algorithm which asymptotically matches the space complexity of the best non-robust tracking algorithm for distinct elements.

Our next set of results provides adversarially robust algorithms for F_p-estimation with $p > 0$. The following result concerns the case $0 < p \leq 2$. It was previously shown that for p bounded away from one, $\Omega(n)$ space is required to deterministically estimate $\|f\|_p^p$, even in the insertion only model [2, 8]. On the other hand, space-efficient non-robust randomized algorithms for F_p-estimation exist. We leverage these, along with an optimized version of the sketch switching technique to save a $\log n$ factor, and obtain an adversarially robust algorithm for F_p-estimation, where $0 < p < 2$.

The next result concerns F_p-estimation for $p > 2$. Here again, we provide an adversarially robust algorithm which is optimal up to a small multiplicative factor. This result applies the computation paths robustification method as a black box. Notably, a classic lower bound of [3] shows that for $p > 2$, $\Omega(n^{1-2/p})$ space is required to estimate $\|f\|_p^p$ up to a constant factor (improved lower bounds have been provided since, e.g., [24, 12]). By using our computation paths technique, we obtain adversarially robust F_p moment estimation algorithms as well for $p > 2$. Lastly, we show that our techniques for F_p moment estimation can be extended to data streams with a bounded number of deletions (negative updates).

Additionally, we show how to get adversarial robust streaming algorithms for a range of problems where it is not clear a-priori how to apply our framework. We show how our techniques can be used to solve the popular *heavy-hitters* problem, and we show how to solve the Entropy estimation problem. See Table 1 for a summary of our results.

Attack on AMS Sketch.

As discussed above, many important streaming problems admit efficient adversarially robust algorithms in the insertion model. It is now natural to ask: are classical algorithms for this problem generally adversarially robust?

We prove that the answer is negative: the classic Alon-Matias-Szegedy sketch (AMS sketch) [2], the first and perhaps most well-known F_2 estimation algorithm (which uses sub-polynomial space), is *not* adversarially robust in the insertion-only model. (In the full turnstile model, in which the adversary is more powerful, the fact that the AMS sketch is not robust follows from the linear sketching lower bound of Hardt and Woodruff [16].) Specifically, we demonstrate an adversary which, when run against the AMS sketch, fools the sketch into outputting a value which is not a $(1\pm\varepsilon)$ estimate of the F_2. The non-robustness of standard static streaming algorithms, even under simple attacks, is a further motivation to design adversarially robust algorithms.

In what follows, recall that the AMS sketch computes $S\cdot f$ throughout the stream, where $S \in \mathbb{R}^{t\times n}$ is a matrix of uniform $\{t^{-1/2}, -t^{-1/2}\}$ random variables. The F_2-estimate is then the value $\|Sf\|_2^2$.

THEOREM 1.2. *Let $S \in \mathbb{R}^{t\times n}$ be the AMS sketch, $1 \leq t \leq n/c$ for some constant $c > 1$. There is an adversary which, with probability $99/100$, succeeds in forcing the estimate $\|Sf\|_2^2$ of the AMS sketch to not be a $(1\pm 1/2)$ approximation of the true norm $\|f\|_2^2$. Moreover, the adversary needs to only make $O(t)$ stream updates before this occurs.*

1.2 Subsequent Work and Open Questions

Based on this paper, a couple of very recent follow-up works have improved upon the space efficiency of our robustification techniques for different settings. Hassidim et al. [18] use techniques from differential privacy to obtain a generic robustification framework in the same mold as ours, where the dependency on the flip number is the improved $\sqrt{\lambda}$ as opposed to linear in λ - the exact bound includes other $\mathrm{poly}((\log n)/\epsilon)$ factors. Similar to our construction, they run multiple independent copies of the static algorithm A with independent randomness and feed the input stream to all of the copies. Unlike our construction, when a query comes, they aggregate the responses from the copies in a way that protects the internal randomness of each of the copies using differential privacy. Using their framework, one may construct an adversarially robust algorithm for F_p-moment estimation that uses $\widetilde{O}(\frac{\log^4 n}{\epsilon^{2.5}})$ bits of memory for any $p \in [0,2]$. This improves over our $\widetilde{O}(\frac{\log n}{\epsilon^3})$ bound for interesting parameter regimes.

Woodruff and Zhou [28] obtain further improvements for a class of problems that have so-called difference estimators which in some cases are (almost) optimal even for the static case. For example, they give an adversarially robust algorithm for F_p-moment estimation that uses $\widetilde{O}(\frac{\log n}{\epsilon^2})$ bits of memory for any $p \in [0,2]$. This improves upon both our work and [18]. Interestingly, difference estimators, which are a new class of algorithms developed in their paper, turn out to be useful also in the sliding windows (classical) model.

Many problems remain open, mainly for achieving optimal bounds for all known streaming problems in the adversarial setting. In particular, one may ask the following:

Do there exist natural streaming tasks that can be solved in the classical setting using small memory, but which require significantly more memory in the adversarial setting?

Very recently, this question was addressed by Kaplan et al. [23] who constructed a streaming problem exhibiting such

a separation between the classical setting, where it only requires a polylogarithmic amount of memory, and the adversarial setting, where polynomial memory is required – that is, an exponential separation. Their construction is based on classical results in adaptive data analysis.

One particular question of interest that remains wide open is related to the turnstile streaming model. The large majority of results in this paper (and in subsequent papers) apply in the insertion-only model. The full turnstile model, where arbitrary insertions and deletions are allowed, is much less understood. In particular we ask the following.

Do there exist small memory streaming algorithms in the adversarial turnstile model for the problems in this paper?

2. TOOLS FOR ROBUSTNESS

In this section, we establish two methods, *sketch switching* and *computation paths*, allowing one to convert an approximation algorithm for any sufficiently well-behaved streaming problem to an adversarially robust one for the same problem. The central definition of a *flip number*, bounds the number of major (multiplicative) changes in the algorithm's output along the stream. As we shall see, a small flip number allows for efficient transformation of non-robust algorithms into robust ones. We remark that the notion of flip number we define here also plays a central role in subsequent works ([18], [28]); for example, the main contribution of the former is a generic robustification technique with an improved (square root type instead of linear) dependence in the flip number. The latter improves the poly$(1/\epsilon)$ dependence on the flip number.

2.1 Flip Number

DEFINITION 2.1 (FLIP NUMBER). *Let $\varepsilon \geq 0$ and $m \in \mathbb{N}$, and let $\bar{y} = (y_0, y_1, \ldots, y_m)$ be any sequence of real numbers. The* ε-flip number $\lambda_\varepsilon(\bar{y})$ *of $\bar{y}$ is the maximum $k \in \mathbb{N}$ for which there exist $0 \leq i_1 < \ldots < i_k \leq m$ so that $y_{i_{j-1}} \notin (1 \pm \varepsilon) y_{i_j}$ for every $j = 2, 3, \ldots, k$.*

Fix a function $g \colon \mathbb{R}^n \to \mathbb{R}$ and a class $\mathcal{C} \subseteq ([n] \times \mathbb{Z})^m$ of stream updates. The (ε, m)-flip number $\lambda_{\varepsilon,m}(g)$ of g over $\mathcal{C}$ is the maximum, over all sequences $((a_1, \Delta_1), \ldots, (a_m, \Delta_m)) \in \mathcal{C}$, of the ε-flip number of the sequence $\bar{=}(y_0, y_1, \ldots, y_m)$ defined by $y_i = g(f^{(i)})$ for any $0 \leq i \leq m$, where as usual $f^{(i)}$ is the frequency vector after stream updates $(a_1, \Delta_1), \ldots, (a_i, \Delta_i)$ (and $f^{(0)}$ is the n-dimensional zeros vector).

The class $\mathcal{C}$ may represent, for instance, the subset of all insertion only streams, or bounded-deletion streams. For the rest of this section, we shall assume $\mathcal{C}$ to be fixed, and consider the flip number of g with respect to this choice of $\mathcal{C}$. We note that a somewhat reminiscent definition, of an *unvarying algorithm*, was studied by [11] (see Definition 5.2 there) in the context of differential privacy. While their definition also refers to a situation where the output undergoes major changes only a few times, both the motivation and the precise technical details of their definition are different from ours.

Note that the flip number is clearly monotone in ε: namely $\lambda_{\varepsilon',m}(g) \geq \lambda_{\varepsilon,m}(g)$ if $\varepsilon' < \varepsilon$. One useful property of the flip number is that it is nicely preserved under approximations. As we show, this can be used to effectively construct approximating sequences whose 0-*flip number* is bounded as a function of the ε-flip number of the original sequence. This is summarized in the following lemma.

LEMMA 2.2. *Fix $0 < \varepsilon < 1$. Suppose that $\bar{u} = (u_0, \ldots, u_m)$, $\bar{v} = (v_0, \ldots, v_m)$, $\bar{w} = (w_0, \ldots, w_m)$ are three sequences of real numbers, satisfying the following:*

- *For any $0 \leq i \leq m$, $v_i = (1 \pm \varepsilon/8) u_i$.*
- *$w_0 = v_0$, and for any $i > 0$, if $w_{i-1} = (1 \pm \varepsilon/2) v_i$ then $w_i = w_{i-1}$, and otherwise $w_i = v_i$.*

Then $w_i = (1 \pm \varepsilon) u_i$ for any $0 \leq i \leq m$, and moreover, $\lambda_0(\bar{w}) \leq \lambda_{\varepsilon/8}(\bar{u})$.

In particular, if (in the language of Definition 2.1) $u_0 = g(f^{(0)}), u_1 = g(f^{(1)}), \ldots, u_m = g(f^{(m)})$ for a sequence of updates $((a_1, \Delta_1), \ldots, (a_m, \Delta_m)) \in \mathcal{C}$, then $\lambda_0(\bar{w}) \leq \lambda_{\varepsilon/8,m}(g)$.

PROOF. The first statement, that $w_i = (1 \pm \varepsilon) u_i$ for any i, follows immediately since $v_i = (1 \pm \varepsilon/8) u_i$ and $w_i = (1 \pm \varepsilon/2) v_i$ and since $\varepsilon < 1$. The third statement follows by definition from the second one. It thus remains to prove that $\lambda_0(\bar{w}) \leq \lambda_{\varepsilon/8}(\bar{u})$.

Let $i_1 = 0$ and let $i_2, i_3, \ldots, i_k$ be the collection of all values $i \in [m]$ for which $w_{i-1} \neq w_i$. Note that $k = \lambda_0(\bar{w})$ and that $v_{i_{j-1}} = w_{i_{j-1}} = w_{i_{j-1}+1} = \cdots = w_{i_j - 1} \neq v_{i_j}$ for any $j = 2, \ldots, k$. We now claim that for every j in this range, $u_{i_{j-1}} \notin (1 \pm \varepsilon/8) u_{i_j}$. This would show that $k \leq \lambda_{\varepsilon/8}(\bar{u})$ and conclude the proof.

Indeed, fixing any such j, we either have $v_{i_{j-1}} > (1 + \varepsilon/2) v_{i_j}$, or $w_{i_{j-1}} < (1 - \varepsilon/2) v_{i_j}$. In the first case (assuming $u_{i_j} \neq 0$, as the case $u_{i_j} = 0$ is trivial),

$$\frac{u_{i_{j-1}}}{u_{i_j}} \geq \frac{v_{i_j}/(1 + \frac{\varepsilon}{8})}{v_{i_{j-1}}/(1 - \frac{\varepsilon}{8})} \geq \left(1 + \frac{\varepsilon}{2}\right) \cdot \frac{1 - \frac{\varepsilon}{8}}{1 + \frac{\varepsilon}{8}} > 1 + \frac{\varepsilon}{8}.$$

In the second case, an analogous computation gives that $u_{i_{j-1}}/u_{i_j} < 1 - \varepsilon/8$. □

Note that the flip number of a function g critically depends on the model in which we work, as the maximum is taken over all sequences of *possible stream updates*; for insertion-only streams, the set of all such sequences is more limited than in the general turnstile model, and correspondingly many streaming problems have much smaller flip number when restricted to the insertion only model. We now give an example of a class of functions with bounded flip number.

PROPOSITION 2.3. *Let $g : \mathbb{R}^n \to \mathbb{R}$ be any monotone function, meaning that $g(x) \geq g(y)$ if $x_i \geq y_i$ for each $i \in [n]$. Assume further that $g(x) \geq T^{-1}$ for all $x > 0$, and $g(M \cdot \vec{1}) \leq T$, where M is a bound on the entries of the frequency vector and $\vec{1}$ is the all 1's vector. Then the flip number of g in the insertion only streaming model is $\lambda_{\varepsilon,m}(g) = O(\frac{1}{\varepsilon} \log T)$.*

PROOF. Observe that $g(f^{(0)}) = 0$, $g(f^{(1)}) \geq T^{-1}$, and $g(f^{(m)}) \leq g(\vec{1} \cdot M) \leq T$. Since the stream has only positive updates, $g(f^{(0)}) \leq g(f^{(1)}) \leq \cdots \leq g(f^{(m)})$. Let $y_1, \ldots, y_k \in [m]$ be any set of points such that $g(f^{(y_i)}) < (1+\varepsilon) g(f^{(y_{i+1})})$ for each i. Since there are at most $O(\frac{1}{\varepsilon} \log T)$ powers of $(1 + \varepsilon)$ between T^{-1} and T, by the pigeonhole principle if $k > \frac{C}{\varepsilon} \log(T)$ for a sufficiently large constant C, then at least two values must satisfy $(1 + \varepsilon)^j \leq g(f^{(y_i)}) \leq g(f^{(y_{i+1})}) \leq (1 + \varepsilon)^{j+1}$ for some j, which is a contradiction. □

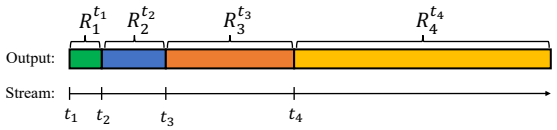


Figure 1: The *sketch switching* method, one of our techniques for transforming a streaming algorithm into an adversarially robust algorithm. As we prove, the following strategy is useful for efficiently robustifying streaming algorithms in a wide range of contexts: maintain several copies $R_1, R_2, \ldots$ of the algorithm, but at any given time t, only communicate to the adaptive adversary the output $R_i^{t_i}$ of a single "active" copy R_i, where $t_i < t$ is the step where R_i became active. We then *switch* the active sketch from R_i to R_{i+1} on the time step t_{i+1} in which the value of $R_i^{t_{i+1}}$ diverges significantly from $R_i^{t_i}$. This will ensure that the algorithm always returns a good approximation, without leaking any meaningful information about its internal state to the adversary.

Note that a special case of the above are the F_p moments of a data stream. Recall here $\|x\|_0 = |\{i : x_i \neq 0\}|$ is the number of non-zero elements in a vector x.

COROLLARY 2.4. *Let $p > 0$. The (ε, m)-flip number of $\|x\|_p^p$ in the insertion only streaming model is $\lambda_{\varepsilon,m}(\|\cdot\|_p^p) = O(\frac{1+p}{\varepsilon}\log m)$.*

PROOF. We have $\|\vec{0}\|_p^p = 0$, $\|z\|_p^p \geq 1$ for any non-zero $z \in \mathbb{Z}$, and $\|f^{(m)}\|_p^p \leq M^p n \leq n^{cp}$ for some constant c, where the second to last inequality holds because $\|f\|_\infty \leq M$ for some $M = \text{poly}(n)$ is assumed at all points in the streaming model. Moreover, for $p = 0$ we have $\|f^{(m)}\|_0 \leq n$. The result then follows from applying Proposition 2.3 with $T = n^{c\cdot\max\{p,1\}}$. □

Having a small flip number is very useful for robustness, as our next two robustification techniques demonstrate.

2.2 The Sketch Switching Technique

Our first technique is called *sketch switching*, and is described in Algorithm 1. The technique maintains multiple instances of a static strong tracking algorithm, where each time step only one of the instances is "active". The idea is to change the current output of the algorithm very rarely. Specifically, as long as the current output is a good enough multiplicative approximation of the estimate of the active instance, the estimate we give to the adversary does not change, and the current instance remains active. As soon as this approximation guarantee is not satisfied, we update the output given to the adversary, deactivate our current instance, and activate the next one in line. By carefully exposing the randomness of our multiple instances, we show that the strong tracking guarantee (which a priori holds only in the static setting) can be carried into the robust setting. By Lemma 2.2, the required number of instances, corresponding to the 0-flip number of the outputs provided to the adversary, is controlled by the $(\Theta(\varepsilon), m)$-flip number of the problem.

LEMMA 2.5 (SKETCH SWITCHING). *Fix any function $g : \mathbb{R}^n \to \mathbb{R}$ and let A be a streaming algorithm that for any $0 < \varepsilon < 1$ and $\delta > 0$ uses space $L(\varepsilon, \delta)$, and satisfies the $(\varepsilon/8, \delta)$-strong g-tracking property on the frequency vectors $f^{(1)}, \ldots, f^{(m)}$ of any particular fixed stream. Then Algorithm 1 is an adversarially robust algorithm for $(1+\varepsilon)$-approximating $g(f^{(t)})$ at every step $t \in [m]$ with success probability $1 - \delta$, whose space is $O\left(L(\varepsilon/8, \delta/\lambda) \cdot \lambda\right)$, where $\lambda = \lambda_{\varepsilon/8,m}(g)$.*

Algorithm 1: Adversarially Robust g-estimation by Sketch Switching

```
λ ← λ_{ε/8,m}(g)
Initialize independent instances A_1, ..., A_λ of
  (ε/8, δ/λ)-strong g-tracking algorithm
ρ ← 1
g̃ ← g(0⃗)
while new stream update (a_k, Δ_k) do
    Insert update (a_k, Δ_k) into each algorithm
      A_1, ..., A_λ
    y ← current output of A_ρ
    if g̃ ∉ (1 ± ε/2)y then
        g̃ ← y
        ρ ← ρ + 1
    Output estimate g̃
end
```

PROOF. Note that for a fixed randomized algorithm $\mathcal{A}$ we can assume the adversary against $\mathcal{A}$ is deterministic without loss of generality (in our case, $\mathcal{A}$ refers to Algorithm 1). This is because given a randomized adversary and algorithm, if the adversary succeeds with probability greater than δ in fooling the algorithm, then by a simple averaging argument, there must exist a fixing of the random bits of the adversary which fools $\mathcal{A}$ with probability greater than δ over the coin flips of $\mathcal{A}$. Note also here that conditioned on a fixing of the randomness for both the algorithm and adversary, the entire stream and behavior of both parties is fixed.

We thus start by fixing such a string of randomness for the adversary, which makes it deterministic. As a result, suppose that y_i is the output of the streaming algorithm on step i. Then given $y_1, y_2, \ldots, y_k$ and the stream updates $(a_1, \Delta_1), \ldots, (a_k, \Delta_k)$ so far, the next stream update (a_{k+1}, Δ_{k+1}) is deterministically fixed. We stress that the randomness of the algorithm is not fixed at this point; we will gradually reveal it along the proof.

Let $\lambda = \lambda_{\varepsilon/8,m}(g)$ and let $A_1, \ldots, A_\lambda$ be the λ independent instances of an $(\varepsilon/8, \delta/\lambda)$-strong tracking algorithm for g. Since $\delta_0 = \delta/\lambda$, later on we will be able to union bound over the assumption that for all $\rho \in [\lambda]$, A_i satisfies strong tracking on some fixed stream (to be revealed along the proof); the stream corresponding to A_ρ will generally be different than that corresponding to ρ' for $\rho \neq \rho'$.

First, let us fix the randomness of the first instance, A_1. Let $u_1^1, u_2^1, \ldots, u_m^1$ be the updates $u_j^1 = (a_j, \Delta_j)$ that the adversary would make if $\mathcal{A}$ were to output $y_0 = g(\vec{0})$ at every time step, and let $f^{(t),1}$ be the stream vector after updates $u_1^1, \ldots, u_t^1$. Let $A_1(t)$ be the output of algorithm A_1 at time t of the stream $u_1^1, u_2^1, \ldots, u_t^1$. Let $t_1 \in [m]$ be the first time step such that $y_0 \notin (1 \pm \varepsilon/2)A_1(t_1)$, if exists (if not we can set, say, $t_1 = m+1$). At time $t = t_1$, we change our output to $y_1 = A_1(t_1)$. Assuming that A_1 satisfies strong tracking for g with approximation parameter $\varepsilon/8$ with respect to the fixed stream of updates $u_1^1, \ldots, u_m^1$ (which holds with probability

at least $1-\delta/\lambda$), we know that $A_1(t) = (1\pm\varepsilon/8)g(f^{(t)})$ for each $t < t_1$ and that $y_0 = (1\pm\varepsilon/2)A_1(t)$. Thus, by the first part of Lemma 2.2, $y_0 = (1\pm\varepsilon)g(f^{(t)})$ for any $0 \le t < t_1$. Furthermore, by the strong tracking, at time $t = t_1$ the output we provide $y_1 = A_1(t_1)$ is a $(1\pm\varepsilon/8)$-approximation of the desired value $g(f^{(t_1)})$.

At this point, $\mathcal{A}$ "switches" to the instance A_2, and presents y_1 as its output as long as $y_1 = (1\pm\varepsilon/2)A_2(t)$. Recall that randomness of the adversary is already fixed, and consider the sequence of updates obtained by concatenating $u^1_1, \ldots, u^1_{t_1}$ as defined above (these are the updates already sent by the adversary) with the sequence $u^2_{t_1+1}, \ldots, u^2_m$ to be sent by the adversary if the output from time $t = t_1$ onwards would always be y_1. We condition on the $\varepsilon/8$-strong g-tracking guarantee on A_2 holding for this fixed sequence of updates, noting that this is the point where the randomness of A_2 is revealed. Set $t = t_2$ as the first value of t (if exists) for which $A_2(t) = (1\pm\varepsilon/2)y_1$ does not hold. We now have, similarly to above, $y_1 = (1\pm\varepsilon)g(f^{(t)})$ for any $t_1 \le t < t_2$, and $y_2 = (1\pm\varepsilon/8)g(f^{(t_2)})$.

The same reasoning can be applied inductively for A_ρ, for any $\rho \in [\lambda]$, to get that (provided $\varepsilon/8$-strong g-tracking holds for A_ρ) at any given time, the current output we provide to the adversary y_ρ is within a $(1\pm\varepsilon)$-multiplicative factor of the correct output for any of the time steps $t = t_\rho, t_\rho + 1, \ldots, \min\{t_{\rho+1}-1, m\}$. Taking a union bound, we get that with probability at least $1-\delta$, all instances provide $\varepsilon/8$-tracking (each for its respective fixed sequence), yielding the desired $(1\pm\varepsilon)$-approximation of our algorithm.

It remains to verify that this strategy will succeed in handling all m elements of the stream (and will not exhaust its pool of algorithm instances before then). Indeed, this follows immediately from Lemma 2.2 applied with $\bar{u} = ((g(f^{(0)}), \ldots, g(f^{(m)})), \bar{v} = (g(f^{(0)}), A_1(1), \ldots, A_1(t_1), A_2(t_1+1), \ldots, A_2(t_2), \ldots)$, and $\bar{w}$ being the output that our algorithm $\mathcal{A}$ provides ($y_0 = g(f^{(0)})$ until time $t_1 - 1$, then y_1 until time $t_2 - 1$, and so on). Observe that indeed $\bar{w}$ was generated from v exactly as described in the statement of Lemma 2.2. □

2.3 The Computation Paths Technique

With our sketch switching technique, we showed that maintaining multiple instances of a non-robust algorithm to estimate a function g, and switching between them when the rounded output changes, is a recipe for a robust algorithm to estimate g. We next provide another recipe, which keeps only one instance, whose success probability for any fixed stream is very high; it relies on the fact that if the flip number is small, then the total number of fixed streams that we should need to handle is also relatively small, and we will be able to union bound over all of them. Specifically, we show that any non-robust algorithm for a function with bounded flip number can be modified into an adversarially robust one by setting the failure probability δ small enough.

LEMMA 2.6 (COMPUTATION PATHS). *Fix $g\colon \mathbb{R}^n \to \mathbb{R}$ and suppose that the output of g uses $\log T$ bits of precision. Let A be a streaming algorithm that for any $\varepsilon, \delta > 0$ satisfies the (ε,δ)-strong g-tracking property on the frequency vectors $f^{(1)}, \ldots, f^{(m)}$ of any particular fixed stream. Then there is a streaming algorithm A' satisfying the following.*

1. *A' is an adversarially robust algorithm for $(1+\varepsilon)$-approximating $g(f^{(t)})$ in all steps $t \in [m]$, with success probability $1-\delta$.*
2. *The space complexity and running time of A' as above (with parameters ε and δ) are of the same order as the space and time of running A in the static setting with parameters $\varepsilon/8$ and $\delta_0 = \delta/\left(\binom{m}{\lambda}T^{O(\lambda)}\right)$, where $\lambda = \lambda_{\varepsilon/8,m}(g)$.*

PROOF. The algorithm A' that we construct runs by emulating A with the above parameters, and assuming that the output sequence of the emulated A up to the current time t is $v_0, \ldots, v_t$, it generates w_t in exactly the way described in Lemma 2.2: set $w_0 = v_0$, and for any $i > 0$, if $w_{i-1} \in (1\pm\varepsilon/2)v_i$ then $w_i = w_{i-1}$, and otherwise $w_i = v_i$. The output provided to the adversary at time t would then be w_t.

As in the proof of Lemma 2.5, we may assume the adversary to be deterministic. This means, in particular, that the output sequence we provide to the adversary fully determines its stream of updates $(a_1, \Delta_1), \ldots, (a_m, \Delta_m)$. Take $\lambda = \lambda_{\varepsilon/8,m}(g)$. Consider the collection of all possible output sequences (with $\log T$ bits of precision) whose 0-flip number is at most λ, and note that the number of such sequences is at most $\binom{m}{\lambda}T^{O(\lambda)}$. Each output sequence as above uniquely determines a corresponding stream of updates for the deterministic adversary; let $\mathcal{S}$ be the collection of all such streams.

Pick $\delta_0 = \delta/|\mathcal{S}|$. Taking a union bound, we conclude that with probability $1-\delta$, A (instantiated with parameters $\varepsilon/8$ and δ_0) provides an $\varepsilon/8$-strong g-tracking guarantee for all streams in $\mathcal{S}$. We fix the randomness of A, and assume this event holds.

At this point, the randomness of both parties has been revealed, which determines an output sequence $v_0, \ldots, v_m$ for the emulated A and the edited output, $w_0, \ldots, w_m$, that our algorithm A' provided to the adversary. The proof now follows by induction over the number t of stream updates that have been seen. The inductive statement is the following:

1. The sequence of outputs that the emulated algorithm A generates in response to the stream updates up to time t, $v_0, \ldots, v_t$, is a $(1\pm\varepsilon/8)$-approximation of g over the stream up to that time.

2. The sequence of outputs that the adversary receives from A' until time t, $(w_0, \ldots, w_t)$, has 0-flip number at most λ (and is a prefix of a sequence in $\mathcal{S}$).

The base case, $t = 0$, is obvious; and the induction step follows immediately from Lemma 2.2. □

Acknowledgments

The authors wish to thank Arnold Filtser for invaluable feedback. This work was done in part in the Simons Institute for the Theory of Computing. Part of this work was conducted while Omri Ben-Eliezer was at Tel Aviv University. Rajesh Jayaram and David P. Woodruff are supported by the Office of Naval Research (ONR) grant N00014-18-1-2562, and the National Science Foundation (NSF) under Grant No. CCF-1815840. Eylon Yogev is funded by the ISF grants 484/18, 1789/19, Len Blavatnik and the Blavatnik Foundation, The Blavatnik Interdisciplinary Cyber Research Center at Tel Aviv University, and The Raymond and Beverly Sackler Post-Doctoral Scholarship.

3. REFERENCES

[1] N. Alon, O. Ben-Eliezer, Y. Dagan, S. Moran, M. Naor, and E. Yogev. Adversarial laws of large numbers and optimal regret in online classification. *CoRR*, abs/2101.09054, 2021.

[2] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. *Journal of Computer and System Sciences*, 58(1):137 – 147, 1999.

[3] Z. Bar-Yossef, T. S. Jayram, R. Kumar, and D. Sivakumar. An information statistics approach to data stream and communication complexity. *Journal of Computer and System Sciences*, 68(4):702–732, 2004.

[4] O. Ben-Eliezer and E. Yogev. The adversarial robustness of sampling. In *Proceedings of the 39th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS, pages 49–62. ACM, 2020.

[5] J. Błasiok. Optimal streaming and tracking distinct elements with high probability. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA, pages 2432–2448. SIAM, 2018.

[6] J. Błasiok, J. Ding, and J. Nelson. Continuous monitoring of l_p norms in data streams. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*, APPROX/RANDOM, pages 32:1–32:13, 2017.

[7] V. Braverman, S. R. Chestnut, N. Ivkin, J. Nelson, Z. Wang, and D. P. Woodruff. Bptree: An ℓ_2 heavy hitters algorithm using constant memory. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS, pages 361–376. ACM, 2017.

[8] A. Chakrabarti and S. Kale. Strong fooling sets for multi-player communication with applications to deterministic estimation of stream statistics. In *IEEE 57th Annual Symposium on Foundations of Computer Science*, FOCS, pages 41–50, 2016.

[9] P. Clifford and I. Cosma. A simple sketching algorithm for entropy estimation over streaming data. In *Proceedings of the 16th International Conference on Artificial Intelligence and Statistics*, AISTATS, pages 196–206, 2013.

[10] D. J. DeWitt, J. F. Naughton, D. A. Schneider, and S. Seshadri. Practical skew handling in parallel joins. In *Proceedings of the 18th International Conference on Very Large Data Bases*, VLDB, pages 27–40, 1992.

[11] C. Dwork, M. Naor, T. Pitassi, and G. N. Rothblum. Differential privacy under continual observation. In *Proceedings of the Forty-Second ACM Symposium on Theory of Computing*, STOC, page 715–724. ACM, 2010.

[12] S. Ganguly and D. P. Woodruff. High probability frequency moment sketches. In *45th International Colloquium on Automata, Languages, and Programming*, ICALP, pages 58:1–58:15, 2018.

[13] A. C. Gilbert, B. Hemenway, A. Rudra, M. J. Strauss, and M. Wootters. Recovering simple signals. In *2012 Information Theory and Applications Workshop*, pages 382–391. IEEE, 2012.

[14] A. C. Gilbert, B. Hemenway, M. J. Strauss, D. P. Woodruff, and M. Wootters. Reusable low-error compressive sampling schemes through privacy. In *2012 IEEE Statistical Signal Processing Workshop*, SSP, pages 536–539. IEEE, 2012.

[15] I. J. Good. C332. surprise indexes and p-values. *Journal of Statistical Computation and Simulation*, 32(1–2):90–92, 1989.

[16] M. Hardt and D. P. Woodruff. How robust are linear sketches to adaptive inputs? In *Proceedings of the 45th Annual ACM Symposium on Theory of Computing*, STOC, pages 121–130, 2013.

[17] N. J. Harvey, J. Nelson, and K. Onak. Sketching and streaming entropy via approximation theory. In *49th Annual IEEE Symposium on Foundations of Computer Science*, FOCS, pages 489–498, 2008.

[18] A. Hassidim, H. Kaplan, Y. Mansour, Y. Matias, and U. Stemmer. Adversarially robust streaming algorithms via differential privacy. In *Advances in Neural Information Processing Systems 33*, NeurIPS, 2020.

[19] R. Jayaram and D. P. Woodruff. Data streams with bounded deletions. In *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS, pages 341–354. ACM, 2018.

[20] R. Jayaram and D. P. Woodruff. Towards optimal moment estimation in streaming and distributed models. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*, APPROX/RANDOM, pages 29:1–29:21, 2019.

[21] A. Kamath, E. Price, and D. P. Woodruff. A simple proof of a new set disjointness with applications to data streams, 2020.

[22] D. M. Kane, J. Nelson, and D. P. Woodruff. On the exact space complexity of sketching and streaming small norms. In *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms*, SODA, pages 1161–1178, 2010.

[23] H. Kaplan, Y. Mansour, K. Nissim, and U. Stemmer. Separating adaptive streaming from oblivious streaming. *CoRR*, abs/2101.10836, 2021.

[24] Y. Li and D. P. Woodruff. A tight lower bound for high frequency moment estimation with small error. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*, APPROX/RANDOM, pages 623–638. Springer, 2013.

[25] I. Mironov, M. Naor, and G. Segev. Sketching in adversarial environments. *SIAM Journal on Computing*, 40(6):1845–1870, 2011.

[26] S. Muthukrishnan. Data Streams: Algorithms and Applications. *Foundations and Trends in Theoretical Computer Science*, 1(2):117–236, 2005.

[27] M. Naor and E. Yogev. Bloom filters in adversarial environments. In *Advances in Cryptology - CRYPTO - 35th Annual Cryptology Conference*, pages 565–584, 2015.

[28] D. P. Woodruff and S. Zhou. Tight bounds for adversarially robust streams and sliding windows via difference estimators. *CoRR*, abs/2011.07471, 2020.

Chiller: Contention-centric Transaction Execution and Data Partitioning for Modern Networks (Technical Perspective)

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

Many computing researchers and practitioners may be surprised to find a "research highlight" which innovates on the way to process database transactions. Work in the early 1970s, by Turing winner Jim Gray and others, established a standard set of techniques for transaction management. These remain the basis of most commercial and open-source platforms [1], and they are still taught in university database classes. So why is important research still needed in this topic? The technology environment keeps evolving, and new performance characteristics mean that new algorithms and system designs become appropriate. This perspective will summarise the early work, and point to how the field has continued to progress.

The transaction abstraction: Ongoing research into the mechanisms has not changed the essential properties that users and application programmers depend on. We are dealing with data, stored in one (or more) systems. Often these are SQL database management platforms which support rich predicate-based selection commands, but some could be simpler key-value stores. Several data access operations can be grouped together into a *transaction*; this means that the whole group together acts to perform a single change in the real-world domain which is reflected in the data. For example, a real-world action of hiring an employee can be performed through a transaction with several statements: it checks entries in the References table, inserts a new record in an Employee table, and also modifies information in Managers, Department, and so on. The acronym ACID describes the essential properties: the transaction should be *atomic* (all the changes in data from these operations are performed - in this case we say the transaction has committed, or else none of the changes are evident in the data), *consistent* (the changes collectively should not violate any integrity conditions on the data), *isolated* (there should not be interference between concurrently active transactions), and *durable* (changes made by a committed transaction should not be removed unless another transaction explicitly does so). Being consistent is a responsibility of the application programmer, while the others are to be enforced by the data storage systems. There are various levels of isolation, of which serializability is the strongest, where the transactions appear to run one after another.

The classical implementation techniques (1970s-1990s): The concept of transaction was supported by several implementation techniques for disk-based stores. The main ideas were used in the prototype relational database management systems such as IBM's System R Practice settled around log records to allow durability despite crashes that corrupt in-memory data, and locks to control interleaving of concurrent activity. For uniform outcome of transactions across multiple stores, the two-phase commit protocol became standard. An elegant theory was also established to reason about all these techniques. The rich theory and practice of transaction management, as they emerged from this period, are described in the reference [2].

On-going innovation (2000 and beyond): Just a few of the important topics in recent years are mentioned here. The increases in capacity of main-memory led to exploration of deterministic concurrency control, where the transaction ordering is not determined dynamically as transactions run and are perhaps impacted by the long delay fetching data from disk. Instead, some fixed rules are used to decide which of the transactions (in a small epoch of time) are run first. The very long message latencies, when data is geo-distributed, have led to attempts to combine the processing of replica consistency, lock management, and commit protocols; an alternative approach to this challenge has been a rise of interest in weak isolation and consistency properties.

Chiller: The following research highlight presents a design called Chiller, driven by new high-bandwidth communication technologies that are becoming common in data centers, which the authors show shift where performance bottlenecks occur in the system. They explore the ways lock holding, replica update, and commit processing, all interact in this new hardware environment. A central insight is to identify which data items are contended, and the system re-orders operations so locks on the contended items are held as briefly as possible. To make this work well, they propose new protocols and also show how this changes the decisions that place data items in different machines for optimal performance.

1. REFERENCES

[1] J. M. Hellerstein, M. Stonebraker, and J. R. Hamilton. Architecture of a database system. *Found. Trends Databases*, 1(2):141–259, 2007.

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

Chiller: Contention-centric Transaction Execution and Data Partitioning for Modern Networks

Erfan Zamanian
Brown University
erfanz@alumni.brown.edu

Julian Shun
MIT CSAIL
jshun@mit.edu

Carsten Binnig
TU Darmstadt
cbinnig@cs.tu-darmstadt.de

Tim Kraska
MIT CSAIL
kraska@csail.mit.edu

ABSTRACT

Distributed transactions on high-overhead TCP/IP-based networks were conventionally considered to be prohibitively expensive. In fact, the primary goal of existing partitioning schemes is to minimize the number of cross-partition transactions. However, with the new generation of fast RDMA-enabled networks, this assumption is no longer valid.

In this paper, we first make the case that the new bottleneck which hinders truly scalable transaction processing in modern RDMA-enabled databases is *data contention*, and that optimizing for data contention leads to different partitioning layouts than optimizing for the number of distributed transactions. We then present Chiller, a new approach to data partitioning and transaction execution, which aims to minimize data contention for both local and distributed transactions.

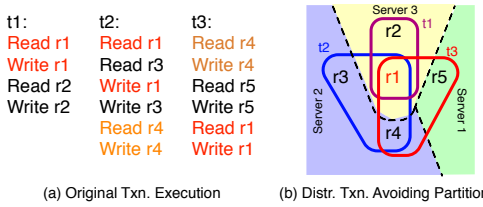
1. INTRODUCTION

The common wisdom is to avoid distributed transactions at almost all costs as they represent the dominating bottleneck in distributed database systems. As a result, many partitioning schemes have been proposed with the goal of minimizing the number of cross-partition transactions (e.g., [2, 10]). Yet, a recent result [16] has shown that with the advances of high-bandwidth RDMA-enabled networks, neither the message overhead nor the network bandwidth are limiting factors anymore, significantly mitigating the scalability issues of traditional systems. This raises the fundamental question of how data should be partitioned across machines given high-bandwidth low-latency networks. We argue that the new optimization goal should be to minimize contention rather than distributed transactions.

In this paper, we present Chiller, a new partitioning scheme and execution model based on two-phase locking which aims to minimize contention. Chiller is based on two complementary ideas: **(1) a novel commit protocol** based on reordering transaction operations with the goal of minimizing the lock duration for contended records through committing such records early, and **(2) contention-aware partitioning** so that the most critical records can be updated without additional coordination, which is different from existing partitioning algorithms that aim to minimize the number of distributed transactions. For example, assume a simple scenario with three servers in which each server can store up to two records, and a workload consisting of three transactions t_1, t_2, and t_3 (Figure 1a). All transactions update r_1. In addition, t_1 updates r_2, t_2 updates r_3 and r_4, and t_3 updates r_4 and r_5. The common wisdom would dictate partitioning the data in a way that the number of cross-cutting transactions is minimized; in our example, this would mean co-locating all data for t_1 on a single server as shown in Figure 1b, and having distributed transactions for t_2 and t_3.

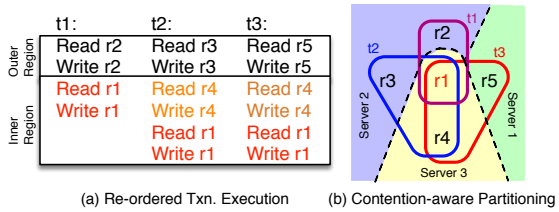
However, as shown in Figure 2a, if we re-order each transaction's operations such that the updates to the most contended items (r_1 and r_4) are done last, we argue that it is better to place r_1 and r_4 on the same machine, as in Figure 2b. At first this might seem counter-intuitive as it increases the total number of distributed transactions. However, this partitioning scheme decreases the likelihood of conflicts and therefore increases the total transaction throughput. The idea is that re-ordering the transaction operations minimizes the lock duration for the "hot" items. More importantly, after the re-ordering, the transaction commit relies entirely on the success of acquiring the lock for the most contended records. That is, if a distributed transaction has already acquired the locks for all non-contended records (referred to as the *outer region*), the commit outcome will only depend on the contended records (referred to as the *inner region*). This allows us to make all updates to the records in the *inner region* without any further coordination. Note that this partitioning technique primarily targets high-bandwidth low-latency networks, which mitigates the two most common bottlenecks for distributed transactions: message overhead and limited network bandwidth.

To provide such a contention-aware scheme, Chiller is based on two complementary ideas that go hand-in-hand: a contention-aware data partitioning algorithm and an operation-reordering execution scheme. First, different from existing partitioning algorithms that aim to minimize the number of distributed transactions (such as Schism [2]), Chiller's partitioning algorithm explicitly takes record contention into account to co-locate hot records. Second, at runtime, Chiller uses a novel execution scheme which goes beyond existing work on re-ordering operations (e.g., QURO [15]). By taking advantage of the co-location of hot records, Chiller's execution scheme reorders operations such that it can release locks



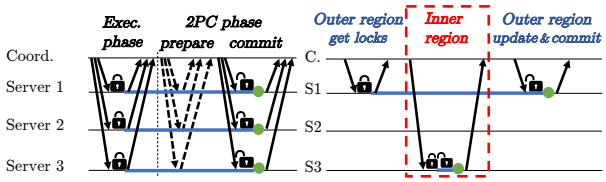
(a) Original Txn. Execution (b) Distr. Txn. Avoiding Partitioning

Figure 1: Traditional Execution and Partitioning.



(a) Re-ordered Txn. Execution (b) Contention-aware Partitioning

Figure 2: Chiller Execution and Partitioning.



(a) 2PC + 2PL (b) Two-region execution

Figure 3: The lifetime of a distributed transaction. The green dots denote when each server releases its locks. The blue lines represent the contention span for each server.

on hot records early and thus reduce the overall contention span on those records.

In summary, we make the following contributions:

(**1**) We propose a new contention-centric partitioning scheme.
(**2**) We present a new distributed transaction execution technique, which aims to update highly-contended records without additional coordination.
(**3**) We show that Chiller outperforms existing techniques by up to a factor of 2 on various workloads.

2. OVERVIEW

2.1 Transaction Processing with 2PL & 2PC

To understand the impact of contention in distributed transactions, let us consider a traditional two-phase locking (2PL) protocol with two-phase commit (2PC). Here, we use transaction t_3 from Figure 1, and further assume that its coordinator is on Server 1, as shown in Figure 3a. The green circle on each partition's timeline shows when it releases its locks and commits. We refer to the time span between acquisition and release of a record lock as the record's *contention span* (depicted by thick blue lines), during which all concurrent accesses to the record would be conflicting. In this example, the contention span for all records is 2 messages long with piggybacking optimization (when merging the last step of execution with the prepare phase) and 4 without it.

2.2 Contention-Aware Transactions

We propose a new partition and execution scheme that aims to minimize the contention span for contended records. The partitioning layout shown in Figure 2b opens new possibilities. As shown in Figure 3b, the coordinator requests locks for all the non-contended records in t_3, which is r_5. If successful, it will send a request to the partition hosting the hot records, Server 3, to perform the remaining part of the transaction. Server 3 will attempt to acquire the lock for its two records, complete the read-set, and perform the transaction logic to check if the transaction can commit. If so, it **commits** the changes to its records.

The reason that Server 3 can unilaterally commit or abort before the other involved partitions receive the commit decision is that Server 3 contains all necessary data to perform the transaction logic. Therefore, the part of the transaction which deals with the hottest records is treated as if it were an independent *local* transaction. This effectively makes the contention span of r_1 and r_4 much shorter (just local memory access, as opposed to at least one network roundtrip).

2.3 Discussion

There are multiple details hidden in the execution scheme presented above. First, after sending the request to Server 3, neither the coordinator nor the other partitions is allowed to abort the transaction; this decision is only up to Server 3. This requirement is very similar to that of H-Store [6], VoltDB [12], and Calvin [13].

Second, for a given transaction, the number of partitions for the inner region has to be *at most* one. Otherwise, multiple partitions cannot commit independently without coordination. This is why executing transactions in this manner requires a new partitioning scheme to ensure that contended records that are likely to be accessed together are co-located.

Finally, our execution model needs to have access to the transaction logic in its entirety to be able to re-order its operations. Our prototype achieves this by running transactions through invoking stored procedures, though it can be realized by other means such as implementing it as a query compiler (similar to Quro [15]). The main alternative model, namely interactive transactions, in which there may be multiple back-and-forth rounds of network communication between a client application and the database is extremely unsuitable for applications that deal with contended data, because all locks and latches have to be held for the entire scope of the client interaction [14].

3. TWO-REGION EXECUTION

The goal of the two-region execution is to minimize the duration of locks on contended records. To achieve this, the execution engine re-orders operations into cold (outer region) and hot operations (inner region); the outer region is executed as normal. If successful, the records in the inner region are accessed. The inner region *commits* upon completion without coordinating with the other participants.

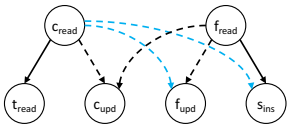
To explain the concepts, we will use an imaginary flight-booking transaction shown in Figure 4a. Here, there are four tables: flight, customer, tax, and seats. In this example, if the customer has enough balance and the flight has an available seat (line 12), a seat is booked and the ticket fee plus state-tax is deducted from their account (lines 14–16). Otherwise, the transaction aborts (line 19).

3.1 Constructing a Dependency Graph

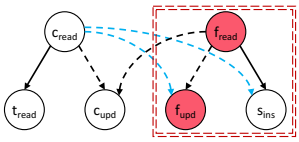
We now describe how we extract the constraints in reordering operations from the transaction logic and model it as a dependency graph.

```
1  // input: flight_id, cust_id
2  // desc: reserve a seat in the flight
3           and deduct the ticket fee from
4           customer's balance.
5  Begin transaction
6    f = read("flight", key:flight_id)
7    c = read("customer", key:cust_id)
8    t = read("tax", key:c.state)
9
10   cost = calculate_cost(f.price, t)
11
12   if (c.balance >= cost AND f.seats > 0){
13     seat_id = f.seats
14     update(f, f.seats ← f.seats - 1)
15     update(c, c.balance ← c.balance - cost)
16     insert("seats", key:[flight_id, seat_id],
17                     value: [cust_id, c.name])
18   }
19   else abort
20 End transaction
```

(a) Original transaction



(b) Static analysis: Construct dependency graph



(c) Step 1&2: Select the inner host, if exists

```
// input: cust_id
Begin Outer Region – Phase 1
  c = read_with_wl("customer", key:cust_id)
  t = read_with_rl("tax", key:c.state)
End Outer Region – Phase 1
```

(d) Step 3: Read records in the outer region

```
// input: flight_id, tax t, customer c
Begin Inner Region
  f = read("flight", key:flight_id)
  cost = calculate_cost(f.price, t)
  if (c.balance >= cost AND f.seats > 0)
    seat_id = f.seats
    update(f, f.seats ← f.seats - 1)
    insert("seats", key:[flight_id, seat_id])
  else abort
End Inner Region
```

(e) Step 4: Execute and "**commit**" the inner region

```
// input: customer c, cost
Begin Outer Region – Phase 2
  update(c, c.balance ← c.balance - cost)
End Outer Region – Phase 2
```

(f) Step 5: Commit the outer region

Figure 4: Two-region execution of a ticket purchasing transaction. In the dependency graph, primary key and value dependencies are shown in solid and dashed lines, respectively (blue for conditional constraints, e.g., an "if" statement). Assuming that the flight record is contended (red circles), the red box in (c) shows the operations in the inner region (Step 4). The rest of the operations will be performed in the outer region (Steps 3 and 5).

There may be constraints on data values that must hold true (e.g., seat availability). Furthermore, operations in a transaction may have dependencies among each other. The goal is to reflect such constraints in the *dependency graph*. Here, we distinguish between two types of dependencies. A *primary key dependency (pk-dep)* is when accessing a record r_2 can happen only after accessing record r_1, as the primary key of r_2 is only known after r_1 is read (e.g., the read operation for the tax record in line 8 must happen after the read operation for the customer record on line 7). In a *value dependency (v-dep)*, the new values for a record r_2 are known only after accessing r_1 (e.g., the update operation on line 15). pk-deps determine the constraints in re-ordering operations, while v-deps do not.

Each operation corresponds to a node in the dependency graph. There is an edge from node n_1 to n_2 if the corresponding operation of n_2 depends on that of n_1. The dependency graph for our running example is shown in Figure 4b. For example, the insert operation on line 16 (s_{ins} in the graph) has a pk-dep on the read operation on line 6 (f_{read}), and has a v-dep on the read operation on line 7 (c_{read}). This means that obtaining the lock for the insert query can only happen after the flight record is read (pk-dep), but can happen before the customer is read (v-dep).

3.2 Run-Time Decision

Given the dependency graph, we describe step by step how the protocol executes a two-region transaction.

1) Decide on the execution model: First, the algorithm finds the candidate operations for the inner region. An operation can be a candidate if the records accessed by it are marked as contended in the lookup table, and it does not have any pk-dep to other partitions, since if it does, it would make early commit of the inner region impossible. In Figure 4b, if the insert operation s_{ins} belongs to a different partition than f_{read}, the latter cannot be considered for the inner region because there is a pk-dep between them.

Finding the hosting partition of an operation which accesses records by their primary keys is quite straightforward. However, finding this information for operations which access records by non-primary-key attributes may require secondary indexes. In case no such information is available, such operations will not be considered for the inner region.

2) Select the host for inner region: If all candidate operations for the inner region belong to the same host, then it is chosen as the *inner host*, and otherwise, a single host has to be chosen. Currently, we choose the host with the highest number of candidate operations as the inner host.

3) Read records in outer region: The transaction attempts to lock and read the records in its outer region. In our example, an exclusive lock for the customer record and a shared lock for the tax record are acquired. If either of these lock requests fails, the transaction aborts.

4) Execute and commit inner region: Once all locks are acquired for the outer region, the coordinator delegates processing the inner region to the inner host by sending a message with all information needed to execute its part. Having the values for all of the records in the read-set allows the inner host to check if all of the constraints in the transaction are met (e.g., that there are free seats in the flight). This guarantees that if an operation in the outer region results in an abort, it will be detected by the inner host and the entire transaction will abort.

Once all locks are acquired and the transaction logic is checked to ensure that it can commit, the inner host updates the records, replicates its changes to its replicas (Section 5.1), and *commits*. In case any of the lock requests or transaction constraints fails, the inner host aborts the transaction and directly informs the coordinator about its decision. In our example, the update to the flight record is applied, a new record gets inserted into the seats table, the partial transaction commits, and the value for the `cost` variable is returned, as it will be needed to update the customer's balance.

5) Commit outer region: If the inner region succeeds, the transaction is already considered committed and the coordinator must commit all changes in the outer region. In our example, the customer's balance is updated, and the locks are released from the tax and customer records.

3.3 The Need for a New Partitioning

The two-region execution will not be useful if the transaction's hot records reside in different partitions. No matter which partition becomes the inner host, the contended

records on the other partitions will observe long contention spans. Therefore, frequently co-accessed hot records must be co-located per transaction. To this end, we present a novel partitioning technique in Section 4.

4. CONTENTION-AWARE PARTITIONING

To fully unfold the potential of our execution model, Chiller must find contention-minimizing horizontal partitioning of data. We will use the transactions in Figure 5 to explain the partitioning idea. The shade of red corresponds to the record hotness (darker is hotter), and the goal is to find two balanced partitions. Existing partitioning tools (e.g. Schism [2]) minimize distributed transactions (Figure 5b). However, such a split would increase the contention span for the records in transaction t_2, because t_2 would have to hold locks on either 3 or 4, and 6 as part of an outer region.

4.1 Overview of Partitioning

To measure the hotness of records, servers randomly sample the transactions' read- and write-sets during execution. These samples are aggregated over a pre-defined time interval by the *partitioning manager* server (PM). PM uses this information to estimate the contention of individual records (Section 4.2). It then creates the graph representation of the workload, which accommodates the requirements for the two-region execution model (Section 4.3). Based on this representation, it uses a graph partitioning tool to partition the records with the objective of minimizing the overall contention of the workload (Section 4.4). Finally, it updates the servers' lookup tables with new partition assignments.

We assume henceforth that the unit of locking is records. However, the same concepts apply to more coarse-grained lock units, such as pages or hash buckets.

4.2 Contention Likelihood

Using the aggregated samples, PM calculates the conflict likelihood for each record. Due to space constraints, we omit the details and only show the final equation that we derived for calculating record contention. We refer the curious readers to our extended published paper [17]. In the following equation, we use $P_c(\rho)$ to refer to the contention likelihood of record ρ.

$$P_c(X_w, X_r) = 1 - e^{-\lambda_w} - \lambda_w e^{-\lambda_w} e^{-\lambda_r}$$

λ_w and λ_r are time-normalized access frequency for reading and writing to the record, respectively. When λ_w is zero, meaning no writes have been made to the record, the contention will be zero, since shared locks are compatible so no conflict is expected. With a non-zero λ_w, higher values of λ_r will increase the contention likelihood due to the conflict of read and write locks.

4.3 Graph Representation

Chiller models workloads quite differently from existing partitioning algorithms, since record contention must be captured in the graph as minimizing the overall contention is the main objective.

As shown in Figure 5c, we model each transaction as a star; at the center is a dummy vertex (referred to as a *t-vertex*, denoted by a square) with edges to all of the records that are accessed by that transaction. Thus, the number of vertices in the graph is $|T| + |R|$, where $|T|$ is the number of transactions and $|R|$ is the number of records.

All edges connecting a given record-vertex (*r-vertex*) to all of its t-vertex neighbors have the same weight. This weight is proportional to the record's contention likelihood. The weight of the edge between an r-vertex and a connected t-vertex reflects how bad it would be if the record were not accessed in the inner region of that transaction.

Applying the contention likelihood formula to our running example and normalizing the weights produces the graph with the edge weights in Figure 5c. Next, we describe how our partitioning algorithm takes this graph as input and generates a partitioning with low contention.

4.4 Partitioning Algorithm

More formally, our goal is to find a partitioning, which minimizes the contention:

$$\min_S \sum_{\rho \in R} P_c^{(S)}(\rho)$$

$$s.t.\ \forall p \in S : L(p) \leqslant (1+\epsilon) \cdot \mu$$

Here, S is a partitioning of the set of records R into k partitions, $P_c^{(S)}(\rho)$ is the contention likelihood of record ρ under partitioning S, $L(p)$ is the load on partition p, $\mu = \frac{\sum_{p \in P} L(p)}{|P|}$ is the average load on each partition, and ϵ is a small constant that controls the degree of imbalance.

Chiller makes use of METIS [7], a graph partitioning tool which aims to find a high-quality partitioning of the input graph with a small cut, while at the same time respecting the constraint of approximately balanced load across partitions.

The interpretation of the partitioning is as follows: A cut edge e connecting a r-vertex v in one partition to a t-vertex t in another partition implies that t will access v in its outer region, thus observing a conflicting access with a probability proportional to e's weight. To put it differently, the partition to which t is assigned determines t's inner host, and all r-vertices assigned to the same partition can be executed in its inner region. Therefore, a split that minimize the total weight of all cut edges also minimizes the contention.

In our example, the sum of the weights of all cut edges (green lines) is 1.3. Transaction t_1 will access record 3 in its inner region as its t-vertex is in the same partition as record 3, while it will access records 1 and 2 in its outer region. Even though the number of multi-partition transactions is increased compared to Figure 5b, this split results in a much lower contention (1.3 for Chiller as opposed to 3.7 for the partitioning Figure 5b).

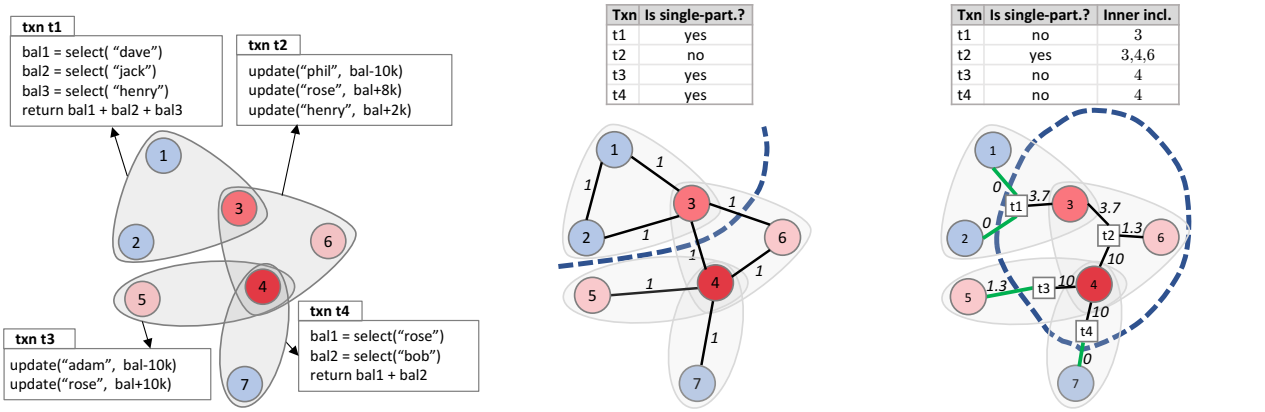
The load L for a partition can be defined in different ways, such as the number of executed transactions, hosting records, or record accesses. The vertex weights depend on the chosen load metric. For the metric of number of executed transactions, t-vertices have a weight of 1 while r-vertices will have a weight of 0.

4.5 Discussion

4.5.1 Scalability of Partitioning.

Chiller has a unique advantage over existing partitioning techniques when it comes to large data sets: it produces graphs with typically significantly fewer edges. Schism, for instance, introduces a total of $n(n-1)/2$ edges for a transaction with n records [2]. However, Chiller's star representation introduces only n edges per transaction, resulting in a much smaller graph. This results in a huge partitioning time improvement.

Furthermore, our approach provides a unique opportunity



(a) The original workload. The hyperedges show the transactions' boundaries. Darker red indicates higher contention.

(b) Distributed transaction minimization schemes. Edge weights are co-access frequencies.

(c) Contention-centric partitioning. Squares denote transactions. Green edges show outer regions.

Figure 5: An example workload and how partitioning techniques with different objectives will partition it into two parts.

to reduce the size of the lookup table. As we are mainly interested in reducing contention, only records whose contention is above a given threshold can be put in the lookup table. Hence, the lookup table needs to store the information for only those hot records. The other records can be partitioned using hash or range functions, which takes no lookup table space. Please refer to our published work to see the benefits of using this technique in action [17].

4.5.2 *Re-Partitioning*

While the process described in Section 4.1 can be done periodically for the purpose of re-partitioning, our current prototype is based on an offline implementation of the Chiller partitioner. We envision that the offline re-partitioning scheme would be sufficient for many workloads. For other workloads with more frequently changing hot spots, however, it is possible that constantly relocating records in an incremental fashion is more effective.

4.5.3 *Minimizing Distributed Transactions.*

In order to co-optimize for contention and distributed transactions, one only needs to assign a minimum positive weight to all edges in the graph. The bigger the minimum weight, the stronger the objective to co-locate records from the same transaction. While co-optimization is still relevant even in fast RDMA-enabled networks due to higher latency of remote access, we argue that minimizing distributed transactions is just a secondary optimization, as the optimal partitioning objective should shift in the direction of minimizing contention.

5. FAULT TOLERANCE

The two-region execution model modifies 2PC for transactions accessing contended records. A transaction is considered committed once its processing is finished by the inner host, after which, it *must* be able to commit on the other participants even under failures. Chiller employs write-ahead logging to non-volatile memory. However, similar to 2PC, while logging enables crash recovery, it does not provide high availability. The failure of the inner host may sacrifice availability, since the coordinator would not know if the inner region has already committed or not. To achieve high availability, Chiller relies on a new replication method based on synchronous log-shipping, described below.

5.1 Replication Protocol

Since in Chiller, the transaction commit point (i.e., when the inner region commits) happens *before* the outer region participants commit their changes, the inner region replication cannot be postponed until the end of the transaction, as otherwise its changes may be lost if the inner host fails.

To solve this problem, Chiller employs two different algorithms for the replication of the inner and outer regions. The changes in the outer region are replicated as normal—once the coordinator finishes performing the operations in the transaction, it replicates the changes to the replicas of the outer region before making the changes visible. The inner region replication, however, must be done *before the transaction commit point*, so that the commit decision will survive failures. Below, we describe the inner region replication in terms of the different roles of the participants:

Inner host: As shown in Figure 6, when the inner host finishes executing its part, it sends an RPC message to its replicas containing the new values of its records, the transaction read-set, and the sequence ID of the replication message. It then waits for the acks from its NIC hardware, which guarantee that the messages have been successfully sent to the replicas. Finally, it safely commits its changes.

Inner host replicas: Each replica applies the updates in the message in the sequence ID order. This guarantees that the data in the replicas synchronously reflect the changes in the primary inner host partition. When updates are applied, each replica notifies the *original coordinator* of the transaction, as opposed to responding back to the inner host.

Coordinator: The coordinator is allowed to resume the transaction only after it has received the notifications from of all the replicas of the inner host.

5.2 Failure Recovery

The recovery procedure is as follows: First, each partition p probes its local log, and compiles a list of pending transactions on p. For each transaction, its coordinator, inner host, and the list of outer region participants are retrieved, and are then aggregated at a designated node to create a global list of pending transactions. Below, we discuss possible failure scenarios for a pending two-region transaction

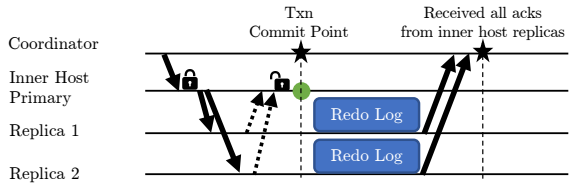


Figure 6: Replication algorithm for the inner region.

along with how the fault tolerance is achieved.

Failure of inner host: If none of the surviving replicas of a failed inner host have received the replication message, the transaction can be safely aborted, because it indicates that the inner host has not committed either. However, if at least one of its replicas has received such a message, that transaction *can* commit, even though that the transaction's updates might have not yet been replicated on all of the replicas. In this case, the coordinator finishes the transaction on the remaining inner host replicas and commits.

Failure of coordinator: If a node is found to be the inner host (or one of its replicas, in case the inner host has also failed), it will be elected as the new coordinator, since it already has the values for the transaction read-set. Otherwise, the transaction can be safely aborted because its changes have not yet received/committed by its inner host.

Failure of an outer region participant: If the failure of participant i happens before the coordinator initiates the inner region, then the transaction is safely aborted. Otherwise, one of i's replicas which has been elected as the new primary will be used to take over i's role in the transaction.

For a sketch of a proof of correctness, we refer the reader to our published work [17].

6. EVALUATION

We evaluated our system to answer two main questions:

(**1**) How does Chiller and its two-region execution model perform under various levels of contention compared to existing techniques?

(**2**) Is the contention-aware data partitioning effective in producing results that can efficiently benefit from the two-region execution model?

6.1 Setup

The test bed we used for our experiments consists of 7 machines connected to a single InfiniBand EDR 4X switch using a Mellanox ConnectX-4 card. Each machine has 256GB RAM and two Intel Xeon E5-2660 v2 processors with 2 sockets and 10 cores per socket. In all experiments, we use only one socket per machine to which the NIC is directly attached. The machines run Ubuntu 14.04 Server Edition as their OS and Mellanox OFED 3.4-1 driver for the network.

6.2 Baselines

We compare the two-region execution scheme against the following commonly used concurrency control (CC) models:

Two-Phase Locking (2PL): we implemented two widely used variants of distributed 2PL with deadlock prevention. In `NO_WAIT`, the system aborts a transaction once it suspects that there is a possibility of deadlock. Therefore, waiting for locks is not allowed. In `WAIT_DIE`, transactions are assigned unique timestamps before execution. An older transaction is allowed to wait for a lock that is owned by a younger transaction, and otherwise it aborts. Timestamp ordering ensures that no deadlock is possible.

Optimistic (OCC): In `OCC`, each participant verifies that its read-set has not been modified by some other transaction. The coordinator commits a transaction only if all the participants pass the validation phase. We based our implementation on the MaaT protocol [8], which is an efficient and scalable algorithm for `OCC` in distributed settings.

In addition, we evaluate two common partitioning schemes: **Hash-partitioning** is the method of assigning records to partitions based on the hash value of their primary key(s). **Schism** is the most notable automatic partitioning technique. It first uses METIS to find a small cut of the workload graph, then compares this record-level partitioning to both a decision tree-learned range partitioning and a simple hash partitioning, and picks the one which results in the minimum number of distributed transactions, or if equal, requires a smaller lookup table. We include the results for different CC schemes for Schism partitioning, and report only `NO_WAIT` for hash partitioning as a simple baseline.

6.3 Workloads

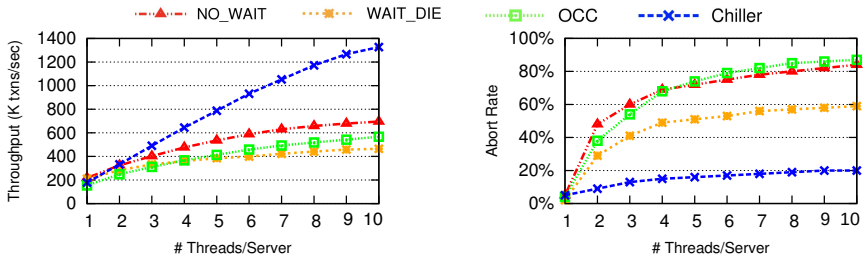
We extensively evaluated Chiller using different workloads. In this paper, we show the results for a subset of our experiments on TPC-C and InstaCart. We refer the reader to our published paper [17] for a more comprehensive experimental evaluation, including more experiments on these two workloads and also various YCSB experiments.

TPC-C: It is the de facto standard for evaluating OLTP systems. Despite being highly partitionable, it contains two severe contention points: the *warehouse* table, and the *district* table. We used one warehouse per server (i.e., 7 warehouses in total) which translates to a high contention workload. As common in all TPC-C evaluations, all tables are partitioned by warehouse ID, except for the Items table which is read-only and is therefore replicated on all servers. Both Chiller and Schism produce this partitioning given the workload trace. Therefore in the following experiments, we mainly focus on the two-region execution feature of Chiller, and evaluate it against the other CC schemes.

InstaCart: To assess the effectiveness of our approach to deal with difficult to partition workloads, we used a real-world data set released by Instacart [4], which is an online grocery delivery service. The dataset contains over 3 million grocery orders for around 50K items from more than 200K customers. On average, each order contains 10 grocery products purchased in one transaction. To model a transactional workload based on the Instacart data, we used the TPC-C's `NewOrder` where each transaction reads the stock values of a number of items, subtracts each one by 1, and inserts a new record in the order table. However, instead of randomly selecting items according to the TPC-C specification, we used the actual Instacart data set. Unlike TPC-C, this data set is actually difficult to partition due to the nature of grocery shopping, where items from different categories (e.g., dairy, produce, and meat) may be purchased together. There is also a significant skew in the number of purchases of different products (e.g. 15% of transactions contain banana).

6.4 TPC-C Results

We first measure the performance of Chiller, `NO_WAIT`, `WAIT_DIE`, and `OCC` with increasing number of worker threads per server. For this experiment, we use TPC-C workload. Although increasing the number threads provides more CPU power to process transactions, it also increases the con-



(a) Throughput. (b) Abort rate.

Figure 7: Comparison of different concurrency control methods and Chiller for TPC-C.

tention. Studying this factor is therefore of great importance since many modern in-memory databases are designed for systems with multi-core CPUs.

As Figure 7a shows, with only one worker thread per machine, `NO_WAIT` and `WAIT_DIE` perform similarly, and have 10% higher throughput than Chiller. This is accounted for by the two-region execution overhead. However, increasing the number of worker threads also raises the chance of conflicts, negatively impacting the scalability of 2PL and `OCC`. Chiller, on the other hand, minimizes the lock duration for the two contention points in TPC-C (warehouse and district records) and thus, scales much better. With 10 threads, the throughput of Chiller is 2× and 3× higher than that of `NO_WAIT` and `WAIT_DIE`, respectively.

Figure 7b shows the corresponding abort rates (averaged over all threads). With more than 4 threads, `OCC`'s abort rate is even higher than `NO_WAIT`, which is attributed to the fact that many transactions are executed to the validation phase and are then forced to abort. Compared to the other techniques, the abort rate of Chiller increases much more slowly as the level of concurrency per server increases. This experiment shows the inherent scalability issue with traditional CC schemes when deployed on multi-core systems, and how Chiller manages to significantly alleviate it.

6.5 InstaCart Results

We analyzed the benefits of combining the Chiller's partitioning scheme with the two-region execution model by using a real-world Instacart workload (as introduced in Section 6.3), which is much harder to partition than TPC-C.

In order to understand the effectiveness of the two-region execution, we compare full Chiller (Chiller) to Chiller partitioning without the two-region execution model (ChP) and Chiller partitioning using Quro* (ChP+Quro*). In contrast to ChP which does not re-order operations, ChP+Quro* re-orders operations using Quro [15], which is a recent contention-reduction technique for centralized database systems. Moreover, we compare full Chiller to two other non-Chiller baselines (Hash-partitioning and Schism-partitioning). For both ChP and ChP+Quro* as well as the non-Chiller baselines (Hash and Schism), we only show the results for a `WAIT_DIE` scheme as it yielded the best throughput compared to `NO_WAIT` and `OCC` for this experiment.

As Figure 8 shows, compared to the Hash-partitioning baseline (black line), both ChP and ChP+Quro* (green and red lines) have significantly higher throughput. We found that this is not because the Chiller partitioning reduces the number of distributed transactions, but rather because contended records which are accessed together are co-located, reducing the cost of aborts. More specifically, if a transaction on contented records needs to be aborted, it only takes

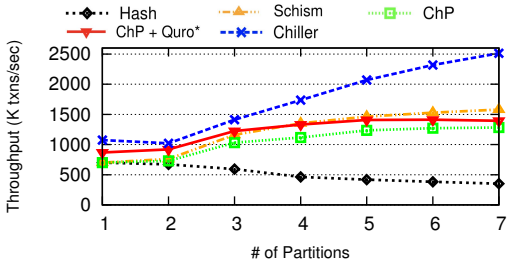


Figure 8: Instacart with different execution models.

one round-trip, leading to an overall higher throughput since the failed transaction can be restarted faster.

Furthermore, we see that ChP+Quro*, which re-orders operations to access the low contended records first, initially increases the throughput by 20% compared to ChP but then its advantage decreases as the number of partitions increases. The reason for this is that the longer latency of multi-partition transactions offsets most of the benefits of operation re-ordering if the commit order of operations remains unchanged. In fact, with 5 partitions, Schism (yellow line) starts to outperform ChP+Quro*, even though Schism does not leverage operation re-ordering.

In contrast to these baselines, Chiller (blue line) not only re-orders operations but also splits them into an inner and outer region, thus outperforms all the other techniques. For the largest cluster size, the throughput of Chiller is approximately 1 million txns/sec higher than the second best baseline. This experiment clearly shows that the contention-centric partitioning must go hand-in-hand with the two-region execution to be most effective.

7. RELATED WORK

Data Partitioning: A large body of work exists for partitioning OLTP workloads with the ultimate goal of minimizing cross-partition transactions. Most notably, Schism [2] is an automatic partitioning tool that uses the workload trace to model the relationship between the database records as a graph, and then applies METIS [7] to find a small cut while approximately balancing the number of records among partitions. Clay [10] builds the same workload graph as Schism, but instead takes an incremental approach to partitioning by building on the previously produced layout. All of these methods share their main objective of minimizing inter-partition transactions. In the age of new networks and much "cheaper" distributed transactions, such an objective is no longer optimal.

Transaction Decomposition: There has also been work on the theory of transaction chopping [11, 18], in which a transaction gets split into smaller pieces, with each piece being

an independent transaction. In contrast to transaction chopping, our two-region execution not only splits a transaction into cold and hot operations, but re-orders operations based on which region they belong to. Also, we do not treat the outer region as an independent transaction and will hold the locks on its records until the end of the transaction. This allows us to abort a transaction later in the inner region.

Determinism and Contention-Reducing Execution: Another line of work aims to reduce contention through enforcing determinism to the concurrency control (CC) unit. Most notably, Calvin [13] uses a global agreement scheme to deterministically sequence the lock requests. Deterministic execution requires *a priori* knowledge of read-set and write-set.

Most related to Chiller is Quro [15], which also re-orders operations inside transactions in a centralized DBMS with 2PL to reduce the lock duration of contended data. However, unlike Chiller, the granularity of contention for Quro is tables, and not records. Furthermore, almost all these works deal with single-node DBMSs and do not have the notion of distributed transactions, 2PC, or asynchronous replication on remote machines, and hence finding a good partitioning scheme is not within their scope.

Transactions over Fast Networks: This paper continues the growing focus on distributed transaction processing on new RDMA-enabled networks [1]. The increasing adoption of these networks by key-value stores [9] and DBMSs [3, 16, 5] is due to their much lower overhead for message processing using RDMA features, low latency, and high bandwidth. The common promise of these systems is better scalability by imposing far less overhead for cross-partition transactions. Therefore, Chiller's two-region execution and its contention-centric partition are specifically suitable for this class of distributed data stores.

8. CONCLUSIONS

This paper presents Chiller, a distributed transaction processing and data partitioning scheme that aims to minimize contention. Chiller is designed for fast RDMA-enabled networks, where the cost of distributed transactions is already low, and the system's scalability depends on the absence of contention in the workload. Chiller partitions the data such that the hot records that are likely to be accessed together are placed in the same partition. Using a novel two-region processing approach, it then executes the *contended* part of a transaction separately from the *un-contended* part. Chiller can significantly outperform existing approaches under workloads with varying degrees of contention.

9. ACKNOWLEDGEMENT

This research is supported by Google, Intel, and Microsoft as part of the MIT Data Systems and AI Lab (DSAIL) at MIT, gifts from Mellanox and Huawei, as well as NSF IIS Career Award 1453171, NSF CAREER Award #CCF-1845763, DOE Early Career Award #DE-SC0018947, and grant BI2011/1 of the German Research Foundation (DFG).

10. REFERENCES

[1] C. Binnig, A. Crotty, A. Galakatos, T. Kraska, and E. Zamanian. The end of slow networks: It's time for a redesign. *PVLDB*, 9(7):528–539, 2016.

[2] C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism: a workload-driven approach to database replication and partitioning. *VLDB Endowment*, 3(1-2):48–57, 2010.

[3] A. Dragojević, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro. No compromises: distributed transactions with consistency, availability, and performance. In *ACM SOSP*, pages 54–70, 2015.

[4] Instacart. The instacart online grocery shopping dataset 2017, 2017.

[5] A. Kalia, M. Kaminsky, and D. G. Andersen. FaSST: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram RPCs. In *12th USENIX OSDI*, pages 185–201, 2016.

[6] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. Jones, S. Madden, M. Stonebraker, Y. Zhang, et al. H-store: a high-performance, distributed main memory transaction processing system. *VLDB Endowment*, 1(2):1496–1499, 2008.

[7] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998.

[8] H. A. Mahmoud, V. Arora, F. Nawab, D. Agrawal, and A. El Abbadi. Maat: Effective and scalable coordination of distributed transactions in the cloud. *VLDB Endowment*, 7(5):329–340, 2014.

[9] C. Mitchell, Y. Geng, and J. Li. Using one-sided RDMA reads to build a fast, cpu-efficient key-value store. In *USENIX Annual Technical Conference*, pages 103–114, 2013.

[10] M. Serafini, R. Taft, A. J. Elmore, A. Pavlo, A. Aboulnaga, and M. Stonebraker. Clay: fine-grained adaptive partitioning for general database schemas. *VLDB Endowment*, 10(4):445–456, 2016.

[11] D. Shasha, F. Llirbat, E. Simon, and P. Valduriez. Transaction chopping: Algorithms and performance studies. *ACM Trans. Database Syst.*, 20(3):325–363, Sept. 1995.

[12] M. Stonebraker and A. Weisberg. The VoltDB main memory DBMS. *IEEE Data Eng. Bull.*, 36(2):21–27, 2013.

[13] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: fast distributed transactions for partitioned database systems. In *ACM SIGMOD*, pages 1–12, 2012.

[14] A. G. Thomson. *Deterministic Transaction Execution in Distributed Database Systems*. PhD thesis, Yale University, 2013.

[15] C. Yan and A. Cheung. Leveraging lock contention to improve OLTP application performance. *VLDB Endowment*, 9(5):444–455, 2016.

[16] E. Zamanian, C. Binnig, T. Harris, and T. Kraska. The end of a myth: Distributed transactions can scale. *VLDB Endowment*, 10(6):685–696, 2017.

[17] E. Zamanian, J. Shun, C. Binnig, and T. Kraska. Chiller: Contention-centric transaction execution and data partitioning for modern networks. In *ACM SIGMOD*, page 511–526, 2020.

[18] Y. Zhang, R. Power, S. Zhou, Y. Sovran, M. K. Aguilera, and J. Li. Transaction chains: Achieving serializability with low latency in geo-distributed storage systems. In *ACM SOSP*, page 276–291, 2013.

Technical Perspective
DIAMetrics: Benchmarking Query Engines at Scale

Peter Boncz
CWI, The Netherlands
boncz@cwi.nl

Benchmarking database systems has a long and successful history in making industrial database systems comparable, and is also a cornerstone of quantifiable experimental data systems research. Creating good benchmarks has been described as something of an art [3]. One can inspire dataset and workload design from "representative" use cases queries, typically informed by domain experts; but also exploit technical insights from database architects in what features, operations, and data distributions should come together in order to invoke a particularly challenging task[1].

While this methodology has served the database community well by creating reference points such as TPC-C and TPC-DS, even in a way creating the narrative on what database workloads are (i.e. transactional vs. analytical), such synthetic benchmarks typically fail to represent actual workloads, which are more complex and thus hard to understand, mixed in nature, and constantly changing. *Automatic benchmark extraction* from real-life workloads therefore provides a powerful pathway towards quantifying database system performance that matters most for an organization, though this requires techniques for summarizing query logs into more compact workloads, and for extracting real data and anonymizing it to create benchmark datasets.

Also, benchmarking in the cloud age needs to deal with complex set-ups in dynamically provisioned environments where distributed compute, storage and network resources need to be orchestrated throughout the benchmarking workflow of dataset creation, loading, workload execution, performance measurement and correctness checking. PEEL[1] has been an earlier effort to facilitate such distributed benchmarking. However, it lacked automatic workload extraction as it focused on executing synthetic benchmarks on Hadoop-based distributed systems running either Spark or Flink.

This paper on DIAMetrics from Deep et al., describes a versatile framework, developed in Google, for automatic extraction of benchmarks and their distributed execution and performance monitoring. The structure of the system is clever, particularly having the components run separately and allowing multiple entry-points into the system to cater to different use cases.

[1]This was coined "choke point"-guided benchmark design by the Linked Data Benchmark Council (ldbcouncil.org)

The observation that input data is not necessarily under the query engines' control, and storage formats may vary, is important. Classic benchmarks assume the engine can preprocess data at will and allows access to detailed statistics about the dataset, which is often not possible in practice.

The state-of-the-art workload extraction approach in DIAMetrics takes into account both structural query features as well as actual execution metrics. This algorithm is described in a separate publication [2].

The data scrambler is a very desirable component, but the paper is light on details here, raising some questions. Certain information leakage is bound to occur, limiting safe use cases. The technique of removing correlations between columns, will also remove important query optimization challenges. This crucial aspect of automatic benchmark extraction seems a fruitful area of future research, e.g. into formal privacy-preservation bounds or privacy-respecting preservation of certain correlations.

At Google, the DIAMetrics framework has proven itself useful for database developers, for performance monitoring and regression testing, as engines evolve. Database users also benefit, using the system to find out which of the multiple Google engines (e.g. F1, Procella, Dremel) best suits their use case, but also to provide performance accountability: to identify and communicate performance problems.

This paper is, in short, a highly recommended read. It inspired me to think about novel directions, possibly taking automatic benchmark extraction from performance monitoring accountability also towards correctness testing: one could envision enriching workload summarization with new dimensions such as code coverage [5] and automatic generation of query correctness oracles [4].

1. REFERENCES

[1] C. Boden, A. Alexandrov, A. Kunft, T. Rabl, and V. Markl. PEEL: A framework for benchmarking distributed systems and algorithms. In *TPCTC 2017*.

[2] S. Deep, A. Gruenheid, P. Koutris, J. F. Naughton, and S. Viglas. Comprehensive and efficient workload compression. *PVLDB*, 14(3):418–430, 2020.

[3] K. Huppler. The art of building a good benchmark. In *TPCTC 2009*.

[4] M. Rigger and Z. Su. Finding bugs in database systems via query partitioning. In *OOPSLA*. ACM, 2020.

[5] R. Zhong, Y. Chen, H. Hu, H. Zhang, W. Lee, and D. Wu. SQUIRREL: testing database management systems with language validity and coverage feedback. In *CCS*, pages 955–970. ACM, 2020.

DIAMetrics: Benchmarking Query Engines at Scale

Shaleen Deep
University of Wisconsin-Madison
shaleen@cs.wisc.edu

Anja Gruenheid
Google Inc.
anjag@google.com

Kruthi Nagaraj
Google Inc.
kruthi@google.com

Hiro Naito
Google Inc.
kiroa@google.com

Jeff Naughton
Google Inc.
naughton@google.com

Stratis Viglas
Google Inc.
sviglas@google.com

ABSTRACT

This paper introduces DIAMetrics: a novel framework for end-to-end benchmarking and performance monitoring of query engines. DIAMetrics consists of a number of components supporting tasks such as automated workload summarization, data anonymization, benchmark execution, monitoring, regression identification, and alerting. The architecture of DIAMetrics is highly modular and supports multiple systems by abstracting their implementation details and relying on common canonical formats and pluggable software drivers. The end result is a powerful unified framework that is capable of supporting every aspect of benchmarking production systems and workloads. DIAMetrics has been developed in Google and is being used to benchmark various internal query engines. In this paper, we give an overview of DIAMetrics and discuss its design and implementation. Furthermore, we provide details about its deployment and example use cases. Given the variety of supported systems and use cases within Google, we argue that its core concepts can be used more widely to enable comparative end-to-end benchmarking in other industrial environments.

1. INTRODUCTION

The data management landscape has drastically changed over the last few years. The majority of database systems are no longer manually tuned and optimized for a specific application by well-versed administrators, instead, they are designed to support a variety of applications. To support all of these applications, a multitude of data models, storage formats and query engines have transformed the data management landscape from standalone, specialized deployments to entire ecosystems. Workloads are now a combination of machine-generated queries for both transactional and analytical workloads as well as ad-hoc queries, varying by application and use case. At the same time, the performance expectations of customers remain the same: They expect the system to be tuned for optimal performance on their workloads. This is commonly achieved in a manual process that first identifies the most important customer use cases which are then used to build curated benchmarks. This process is not principled and may not yield comprehensive benchmarks valid for a long period of time due to (*a*) the dynamic nature of continuously changing production workloads; (*b*) a tight coupling between the workload and underlying query engine, preventing customers from identifying queries that are important across multiple engines; and (*c*) a general lack of understanding of how query performance is affected by small changes to the end-to-end system. Given such complex company-internal ecosystems, it is increasingly difficult to determine for example how well a specific system is performing, how it compares to alternative systems for the same use case, or whether modifying one of its components will negatively impact other parts of the system. However, answering these questions in a principled manner is crucial to companies. DIAMetrics[1] is our answer to this problem setting: a benchmarking framework built at Google with the goals to (*a*) deliver a general solution that is capable of benchmarking end-to-end a variety of query engines; (*b*) support every step of the benchmarking life-cycle; and (*c*) provide insights with respect to system performance and efficiency. It is a one-stop tool for all benchmarking needs including complex tasks such as benchmark generation, execution, and result visualization.

Prior work. Benchmarking data management systems is certainly not new; from the early efforts of the Wisconsin benchmark [2], to the development of industry standards like TPC-H [22], to benchmarks for object-oriented [4] systems, or to larger cloud-scale serving benchmarks [9] and their derivatives. All these benchmarks have been studied extensively and the knowledge gained has been used to modify them in various ways or deliver new benchmarks altogether that address the shortcomings of the existing ones.

The two common aspects of any of these benchmarks have always been that: (*a*) the benchmark workload is statically defined: even if there are randomly seeded data and query generators their outputs all conform to well-defined patterns, i.e., schemas, value distributions, and queries; and (*b*) the system being benchmarked assumes complete control of the entire data management stack, from hardware to software configuration and to manual tuning for optimal performance. Though existing benchmarking efforts certainly serve their purpose for standalone deployments, they are not indicative of production-level data management use cases of an entire ecosystem. There, a query engine does not have control of the data and storage formats; it is expected

 This is a minor revision of the paper entitled DIAMetrics: Benchmarking Query Engines at Scale, published in PVLDB, Vol. 13, No. 12, 3285-3298.
DOI: https://doi.org/10.14778/3415478.3415551

[1]The name stems from the unit within Google DIAMetrics was originally developed to provide metrics for, DIA: Data Infrastructure and Analysis.

to evaluate a wide spectrum of queries, from single-point lookups, to real-time analytics, to extremely large machine-generated queries over a multitude of formats, or any mixture of the above; and it has little to no statistics about the input a priori to guide the system's optimizer and execution engine to deliver robust performance. Static benchmarks can act as a measuring stick, so to speak, but only for the use case they have been designed to address. In all other use cases a static benchmark is often not representative of the actual system load.

Problem motivation. Our work is motivated by the observation that benchmarking is a key necessity to determining the efficiency and usefulness of specific systems for specific tasks. Not having a way to benchmark a production system in a dynamic and often unpredictable environment may prove detrimental not only to the system developer but also to the user. The system developer spends an inordinate amount of time tuning the system for particular use cases and may not have clear insight into the larger-scale problems of the system. For instance, the developer may spend effort optimizing a particular operator at the micro-level, whereas a comprehensive benchmark would have shown that there would be greater benefit optimizing a different part of the system's processing pipeline. Or, the developer may decide that more computing resources are necessary for a particular workload, when a targeted benchmark could showcase that the majority of time is spent on non-compute-intensive execution fragments. The user, on the other hand, benefits from knowing the level of performance a system delivers. For example, if that performance, is suboptimal, she can provide the system developer with examples of this suboptimality, or even move to a different engine that may be better suited to the workload requirements.

Problem solution. Instead of focusing on a specific benchmark workload and using that as the means to test performance and efficiency, we argue that we need a benchmarking framework. That is, an architecture for benchmarking that is capable of generating indicative benchmark workloads over production deployments, executing them, and measuring a system's performance on that workload. Moreover, to avoid duplicate effort, the architecture should be independent of the query engine and it should rely on generic reusable components that can be instantiated with minimal effort for every system that is to be benchmarked. At the same time, the framework should provide the means to track the performance per indicative benchmark workload and use that historical information to measure improvement over time. DIAMetrics provides all that functionality and has been used within Google to benchmark and reason about the end-to-end performance of internal query engines.

While DIAMetrics has only been used within Google, we posit that its architecture is powerful enough to support any query engine, as long as a minimal set of primitives are implemented. This is corroborated by the internal use of DIAMetrics: although Google is a single organization, it exhibits all the diversity characteristics we discussed earlier.

There are at least four internal query engines, each designed for different use-cases: F1 [20, 21], Dremel [17], Spanner SQL [1], and Procella [6]. There exist specialized storage systems such as Mesa [15] and more generic ones such as Colossus [19] which are leveraged by different applications, supporting different storage formats. Figure 1 shows an

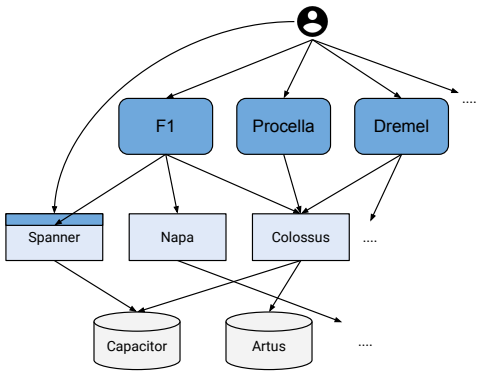


Figure 1: Part of the Google ecosystem.

overview of part of the Google-internal ecosystem of query engines and their dependencies. If every query engine would benchmark according to their own needs, there would be no accountability across engines and no way to determine which systems are useful for which use case. In contrast, DIAMetrics is specifically built to consolidate the benchmarking needs within Google, to provide an effective way to compare engines and at the same time provide means to improve them.

Contributions. In this paper, we present an overview of DIAMetrics, a novel extensible framework for engine-agnostic, repeatable benchmarking that is indicative of large-scale production performance.

Framework Architecture We present the generic architecture of DIAMetrics in Section 2. We show a high-level description of its components and discuss how its design allows for extensions with little effort while seamlessly supporting its core functionalities.

Modular Components Each of the components is highly customizable to cater to customer specific requests while being general enough to handle a variety of use cases.

Use Cases We discuss the deployment and varying use cases as well challenges addressed by the DIAMetrics framework within Google in Section 4.

2. OVERVIEW

DIAMetrics has two primary goals: (*a*) to be fully composable and rely on enhanced reusability in order to facilitate benchmarking at scale; and (*b*) to be able to benchmark and profile any internal system capable of evaluating queries and any customer workload of that system producing these queries. These goals are realized through two key notions:

- *Canonical exchange formats*: for extensive abstraction, whenever two components need to communicate, they do so through well-defined exchange formats that we term *canonical*. The formats are component-dependent, but the intuition is that the module that facilitates the transition from one format to the other can now be 'plugged in': if the component respects these formats all DIAMetrics pipelines remain functional.
- *System drivers*: to interact with all supported query engines, DIAMetrics employs drivers, i.e., modules that are capable of translating canonical workload representations into query processing requests for each

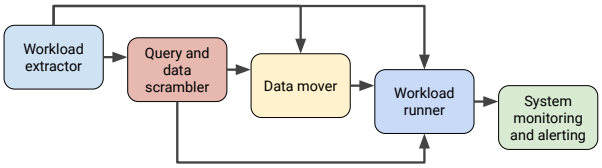


Figure 2: An overview of the DIAMetrics components.

supported query engine, gathering profiling metrics from the execution of that query on the query engine, and translating these metrics to the framework's own canonical profiling format for further processing.

2.1 Components

Using canonical exchange formats and system drivers, we construct the DIAMetrics pipeline as depicted in Figure 2. We use five components when benchmarking a variety of query engines and workloads: (*a*) the workload extractor, (*b*) the query and data scrambler, (*c*) the data mover, (*d*) the workload runner, and (*e*) the system monitoring and alerting component. An overview of these components is given in Section 3. Each component can act as an entry point to the DIAMetrics framework. For instance, some user may not need to create a production workload since they may have one readily available through other means; or they want to use a standard benchmark like TPC-H. Alternatively, another user may only need to test different storage back-ends for the same query workload, so they only need to use the data mover to generate multiple instances of the same workload. Essentially, the components described here are designed in a way that they can be mixed and matched specifically to each benchmarking use case.

2.2 Workflow

Google-internal query engines are highly scalable and are capable of serving billions of queries per day from multiple customers, both internal and external. Each query served, along with a number of internal system-specific information, is logged for example in a distributed logging system running on Colossus, Google's file system [19]. The log formats of each query engine are different, but there is a lot of common information between them stored in different ways. DIAMetrics builds on top of the idea that log entries in essence contain the same information, presented in different ways. Specifically, it uses a canonical representation of a query log, which treats a query as a combination of its query text and a number of features and their values that describe its profile. This representation is leveraged by DIAMetrics to drive the workload extraction and summarization process for custom benchmark generation. In essence, the workload extractor connects to the respective log system, extracting all relevant log entries that may contribute to the benchmark. The summarizer then uses that information to select an optimized subset of these logs that can be used as a custom, representative query workload.

However, these queries are often based on sensitive user data and are thus not available to any outside application or benchmarking system. To address this problem, users can choose to anonymize their data using DIAMetrics's data scrambler. The data scrambler scans the original customer data and applies various anonymization techniques on it in order to ensure no sensitive information is leaked to the benchmark dataset. In the simplest case, the data scrambler will arbitrarily permute the values of a column independently of other columns. Such permutation will ensure that per-column value distributions remain the same, but correlations across columns are broken, thereby reducing the likelihood of disclosure. Additionally, the scrambler may further obfuscate values by hashing them, by mapping them to a different domain, or by adding a small amount of noise so that the resulting dataset has approximately the same statistical properties but over different values; and so on. Finally, depending on the user's benchmarking use case, they might choose to compare their benchmark on a variety of storage layers or with different file formats. The data mover allows DIAMetrics to prepare the benchmark for execution on a variety of back-ends, allowing the user to get a comprehensive understanding of their execution patterns.

Once queries and data are stored in the correct place(s), independent of whether they are derived from the above pipeline or provided by the user, we deploy a workload runner that reads a set of configuration files describing the execution parameters and automatically runs the benchmarks on the specified systems with the specified execution constraints. Following the modular principles explained above, we allow users to write pluggable configurations, i.e., the same system configuration may be used for a set of different benchmarks. Note that by defining these configurations, the user determines the parameters of the benchmark. For example, they can decide to run the benchmark on a production server or in isolation by using different system setups. Similarly, they may choose to compare the generic execution of the TPC-H workload to a platform-optimized version to examine choices made by the query optimizer. In all of DIAMetrics's benchmark executions, we follow standard experimental procedure and allow users to execute the same workload and system configurations multiple times to provide realistic results.

Finally, the last step in the end-to-end workflow is the interpretation of the execution results. The monitoring component of DIAMetrics provides dashboards to the framework users that allow them to easily interpret the historic results of their benchmarks. It is triggered periodically and automatically updates its dashboards whenever new execution results have become available. If desired, users can furthermore use alerts to get notified when their execution patterns change significantly from previously observed or expected patterns. Our end-to-end framework for workload benchmarking has simplified and streamlined benchmarking within Google across different systems. It allows users to set up automatic benchmarking in a matter of minutes without needing to worry about the specific implementation details of executing repeatable benchmarks. In essence, DIAMetrics enables efficient and consistent benchmarking at scale within Google.

3. FRAMEWORK COMPONENTS

We next present the components of the DIAMetrics framework in detail. Each component can be thought of as a stand-alone facility, but it is their interaction that delivers an end-to-end solution.

3.1 Workload extractor

One of the main problems when benchmarking any system is defining the benchmark that appropriately evaluates

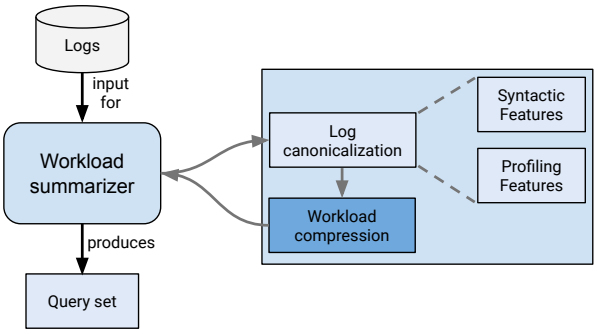


Figure 3: Overview of the workload summarizer.

the system. Indeed, a single system may experience multiple types of workload at different times. For instance, the majority of queries may be long-running resource-intensive analytics queries; or they may be single point lookup queries for record retrieval; or anything in between when a user is using a database in exploratory mode. These can be created by a single or multiple customer(s) using the system for different types of applications. Traditionally, system deployments have been tailored for different application needs, each deployment being optimized for the types of queries it is expected to evaluate. With the move to distributed, large-scale, federated, and cloud-based deployments, however, the advantage of fully controlling the architecture of a system is no longer given. A query engine is treated simply as an end-point and is expected to be able to process user queries with little to no optimization from the user. It is therefore a requirement for the query engine providers to cater to different needs at the same time, which makes it imperative to have a way to gauge the system's performance on the user's workload. Whereas for relational systems we have had benchmarks like TPC-H, TPC-C, or TPC-DS, mostly stemming from the general division of relational workloads into OLTP and OLAP, there are no representative benchmarks for these (user-specific) mixed workloads.

To process not only standardized benchmarks but also user-specific benchmarks, we developed techniques that compress a user's workload into a small set of representative queries that can then be used as a benchmark workload [11]. Our framework for workload extraction and summarization roughly undertakes the following tasks:

Log canonicalization. To create a user-specific benchmark, log entries are first extracted and transformed to a canonical representation that contains a set of *features* necessary to drive summarization. Features can be anything that characterizes the specifics of a query that are deemed useful for benchmark creation. In DIAMETRICS, we support two types of features: syntactic and profiling features. Syntactic features can be extracted by parsing the query, e.g., the number of joins in the query statements or the aggregate functions used in the query. Profiling features on the other hand may encompass characteristics such as query latency, CPU usage or amount of data read/written to disk.

Workload summarization. Once the workload features have been extracted, we can leverage them to identify a subset of queries for benchmarking this workload. The choice of queries in the subset is driven by two metrics: *representativity* and *coverage*. Representativity determines how closely the distribution of features in the subset matches the original workload. In contrast, coverage determines how well the features in the subset cover the features observed in the original workload. To an extent, coverage describes the completeness of the benchmark. During workload summarization, we optimize the selection of queries according to these metrics and greedily pick the benchmark queries which can then be used for realistic production benchmarking. For further details on how to realize workload summarization, please refer to [11].

3.2 Data and query scrambler

In addition to finding a representative set of queries to execute for benchmarking, DIAMETRICS also needs to ensure that the data it is using for these benchmarks is representative. The choice of dataset will drive storage and query processing decisions depending on the query patterns being executed, the storage back-end, the complexity of the data, and the data value distributions, to name but a few factors.

The data scrambler is a step towards addressing the problem of representative data generation, as it provides a simple and efficient way to use production data for query benchmarking. The intent is to have a facility that would allow one to quickly sanitize a representative production dataset and use actual production queries over the sanitized version for performance benchmarking. Once workload summarization identifies the queries that are representative of a workload, we can use the inputs these queries process to snapshot the production data and use that snapshot to build a version of the input data to be used for benchmarking. This is not always straightforward, mainly because production data may contain fields, values and correlations between them that are sensitive and should not be used for benchmarking purposes. In the data scrambler we solve that problem by breaking correlations between values; by protecting data through hashing their values to obfuscate them; and by adding small amounts of noise to the data so that their distributions are not significantly altered. While the scrambler does not provide formal guarantees with respect to privacy or non-disclosure, it has been found to alter the input data in a reasonable way that might be good enough for performance benchmarking in a secure industry setting. No formal guarantees notwithstanding, the scrambler is extremely customizable and may well provide these guarantees implicitly if configured properly by data owners.

3.3 Data mover

The data mover acts as an intermediary between formats. The intuition behind the data mover is to give DIAMETRICS the ability to generate multiple benchmarks from the same workload by converting the same input source to fit different storage back-ends. This is far from trivial as there are multiple aspects to take into account when designing data transformation mechanisms. First, different storage formats imply different schema definitions, which in turn implies potential type conflicts. For instance, the target format supports dates only as milliseconds in the epoch, whereas the format we want to move data from stores these dates as strings; thus, the data mover needs to apply the transformation from one format to another. Second, some input format may have additional statistical information embedded into its sources, or even value indexes incorporated. If that is the case, the data mover deploys a best-effort mechanism to replicate the original input structure with as many auxiliary

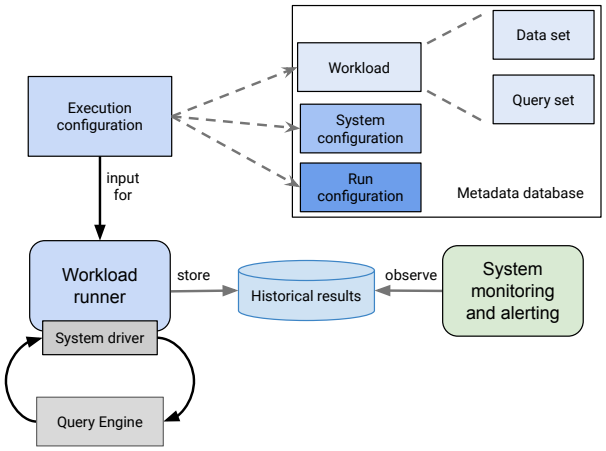


Figure 4: Overview of the workload runner's components.

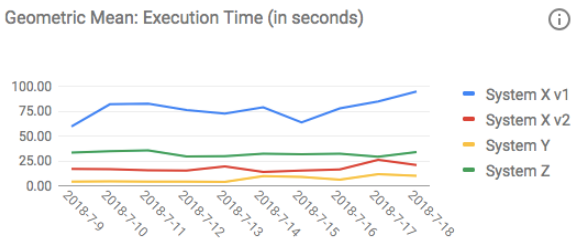
structures transferred to the output as possible. Other information that the data mover attempts to preserve is sharding information, e.g., the number of shards and the partitioning scheme; input storage properties like the input being sorted; data definition properties like functional dependencies if these are supported by the target storage back-end; and, in general, any optimizations that are present in the input dataset and might affect the performance of the storage back-end if they are not preserved. Once the requested data movements have taken place, the input workload will be rewritten so that instead of using the original input sources, it uses the newly generated data sources.

3.4 Workload runner

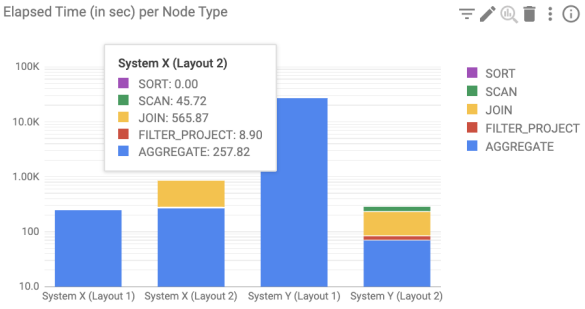
The benchmark execution component of DIAMETRICS is the workload runner. The runner accepts multiple *execution configurations* as input, with each execution configuration containing the following four elements: (*a*) a number of systems and their configurations to use for benchmarking; (*b*) a number of benchmark configurations; (*c*) a number of workloads to benchmark; and (*d*) a number of alert configurations to trigger if there are any issues detected when running a benchmark. The workload runner will then run each execution configuration by deploying every workload over every benchmark configuration and over every system configuration. Configurations are stored in the *metadata database* of DIAMETRICS. When the workload runner is requested to process an execution configuration, it determines the specifics of that configuration in the metadata database, and retrieves all required system, benchmark, workload, and alert configurations it refers to. Next, the runner will deploy an intermediate orchestrator to configure systems and benchmark, run queries, save their profiling metrics, and evaluate any potential alerts as shown in Figure 4. While the default execution is sequential, the workload runner also offers various degrees of parallelism. For example, the runner may send the same workload of all targeted systems for execution in parallel; but within a system it can be configured to issue queries sequentially for better isolation.

3.5 System monitoring and alerting

After the workload runner has executed a workload, we export the output into DIAMETRICS-specific logs. These



(a) Consolidated benchmark execution time tracking



(b) Operation breakdown computation

Figure 5: Example dashboards for per-query benchmark performance monitoring.

logs are then used to (*a*) allow users to monitor benchmarking performance through result visualization, and (*b*) automatically monitor performance regressions and issue alerts. System monitoring is a core objective of DIAMETRICS, as it helps users to track the performance of the system. At the same time, it is useful to system developers to track incremental changes of the same workload, visualizing whether changes to the codebase improved a system's performance. DIAMETRICS automatically retrieves the logs that the workload runner generates and uses the logged profiling metrics for visualization. Specifically, we use static dashboards to visualize the workload execution over time in terms of essential statistics such as latency, CPU time, spilled bytes and any other metric that is captured by the executed system and deemed important by the client. An example of performance tracking of execution time using the same benchmark for various systems is shown in Figure 5a. Here, we observe the execution of three different systems, one of which is executed with two different system settings. Their performance is tracked over a timespan of ten days and the average execution time is reported. We primarily monitor aggregate metrics like the geometric mean of Figure 5a for latency, but this is not the only capability of the monitoring substrate of DIAMETRICS. The dashboard visualizations allow for an intuitive comparison of the different systems and, at the very least, signal (*a*) how stable a system is, and (*b*) how a system fares in comparison to alternative deployments of the same system, or other internal query engines.

In addition to simple tracking dashboards, we also developed more insightful dashboards, such as dashboards that look at the scalability of a system or give insights into specific queries. For example, in Figure 5b, we visualize two different systems that run the same query on two different

data layouts to compare performance. We observe that System X performs comparatively better on data layout 1 while System Y is preferable on data layout 2. Furthermore, these different data layouts lead to different utilization of SQL operators: While data layout 1 results in query execution being dominated by aggregation operations, data layout 2 results in join operations also taking a comparative fraction of query execution. This type of information is invaluable when evaluating different systems as well as storage layers, optimization mechanisms, and so on.

Finally, DIAMetrics can signal to developers and workload owners if there exists a significant performance degradation in the most recent snapshot of the system, if there were any failures, and so on. At an abstract level, the alerting framework uses the metrics produced by the latest run of an execution configuration and compares them to their historical behavior to identify potential regressions. Overall, alerts are an important facility of DIAMetrics in order to identify any deviations from the expected norm and focus the attention of the development, production, and benchmark-owning teams to problematic situations that can be exemplified through a handful of queries exhibiting the problem.

4. DEPLOYMENT & LESSONS LEARNED

DIAMetrics is capable of benchmarking all production-ready generally-available SQL engines within Google as well as selected internal, non-SQL engines. With every workload run it evaluates thousands of queries across multiple systems, gathering and storing performance metrics for immediate and later analysis. It has enabled query engine production teams to set performance goals and know where they stand with respect to alternative systems, while it has also helped teams to migrate between query engines by identifying problematic cases and setting up roadmaps for the migration. We next sketch the most widely encountered use cases for DIAMetrics, some of them expected, but others being off the beaten track with respect to its original design.

4.1 Benchmarking

The core idea behind DIAMetrics is to provide users with an intuitive means to benchmark their systems. To that effect, we have developed DIAMetrics to be modular, customizing the benchmarking experience to the team's use case. Tooling for daily benchmarking is currently used by a variety of Google query engines such as F1, Procella, and Dremel. Some of their use cases are described here:

Workload characterization. One of the highest barriers of entry to DIAMetrics has been that teams often do not have a clear grasp on what their query workload looks like; or, if they know all the query patterns that they employ, they have no means to identify important patterns. In some cases, a simple frequency-based clustering of queries is enough to identify a rough approximation of the workload; but in the majority of cases that is not possible. Workload summarization is a powerful method to compress a workload into a benchmark, providing guarantees about the output in terms of its representativity and coverage. Moreover, the summarizer is capable of delivering the benchmark workload under specific constraints in terms of the profile of the extracted benchmark. Having such a facility in place allows teams to quickly turn their workload into a benchmark with minimal manual log mining and configuration on their part.

Workload optimization. DIAMetrics is capable of producing tracking dashboards for various combinations of system configurations over different versions of the same workload. Internal teams have used DIAMetrics to test their optimizer's performance by comparing out-of-the-box and manually optimized versions of a workload; or to compare the performance of different storage configurations; or to measure the impact of a feature upon a workload by comparing system performance with the feature being turned on or off. Having the ability to do this with minimal configuration and over production workloads in addition to standard benchmarks, improves the confidence of development teams in their decisions and not only optimizes specific use cases, but also reduces the management and financial cost of deploying these workloads. Moreover, doing so on a compressed, representative version of a workload with robustness guarantees allows the data owners to quickly perform these optimizations at scale and extrapolate from the performance of the compressed workload to the expected performance of the system on the actual workload.

Performance accountability. Tracking the historical performance of a query engine on a workload is a two-way street. Not only is it useful for a query engine to track how well it performs on a specific workload, it also works in the inverse direction: the developers of an application using a query engine can hold the engine accountable for the performance it delivers on their application. DIAMetrics can be used to deliver compliance benchmarks for service level objectives between data owners and query engine users, and the production team of the query engine. Such accountability bridges the gap between teams and leads to a common understanding of the expected level of performance.

Data anonymization. The DIAMetrics framework enables the use of actual production-like data for benchmarking, thus eschewing the need to come up with synthetic data generators or not being able to benchmark a system with production-like workloads altogether. Often, internal teams have a good idea of benchmark queries, but it is impossible to run these queries over production data as the data contains sensitive user information that only the owning team should be able to access; adding DIAMetrics as a data accessor is simply not an option, nor is it an option to provide access to the data through some other role. The data scrambler can help in these cases as it can reformat the data in various ways and with user-controlled degrees of anonymization. Moreover, scrambling takes place in ways that preserve the input value distributions, thus making the scrambled data a good representation of the original production data.

4.2 Software development

In addition to traditional benchmarking, we also offer support for developers to run their benchmarks on experimental instances. DIAMetrics has improved awareness of how different changes to the codebase impact different query engines and has started to integrate large-scale benchmarking into the developer's workflow.

System comparison and choice. As the implementation of DIAMetrics progressed, a novel use case emerged: helping new teams decide on the most appropriate query engine for their workload. Whereas for well-established teams it is hard to migrate to a new query engine, newer teams do not have such tie-ins. It is therefore possible for a team

to come along with a representative workload and test that workload on the internal query engines that can support it. They can then make an informed decision as to which query engine provides the best support for their workload. Additionally, if they are keen to work with a specific query engine but that engine is not optimized for the workoad, they can provide the engine's developers with example queries where performance suffers.

Performance-driven development. DIAMETRICS has been frequently used to set performance goals for development teams. One of the typical use cases is to identify problematic worklads for a particular query engine and then set a roadmap for implementing improvements for these workloads. Development teams will then use DIAMETRICS to track their performance on those workloads, observing how their modifications improve the system's performance. At the same time, the development team has assurances that newly introduced improvements do not degrade the performance of other workloads.

Release blocking. Monitoring and alerting give rise to the production of compliance tests for the query engines that DIAMETRICS supports. Recall that our framework can target any existing query engine deployment. Some of these deployments may be staging ones, running a version of the system's binary that is different from the official one; most frequently the latter is a release candidate version. By comparing the performance of a benchmark on the current binary with that of the release candidate, teams can identify potential problems before releasing the candidate and block the release in the presence of a potential regression. One of the welcome side-effects of DIAMETRICS is that it exemplifies the regression through a handful of queries in which the regression manifests. By having this information, development teams can quickly start addressing the regression.

5. RELATED WORK

Benchmarking is not a novel problem, especially in the context of data management [2, 3, 5, 9, 10, 12], but has become increasingly important over the last years with the increase in available data, the move to hosted management and data services, and the need for low latency processing regardless of data size. All systems need to be robust, i.e., they need to consistently execute their workloads without performance degradation due to changes in the data or the underlying codebase. Robustness has been discussed in several lines of research in the broader context of database systems. For example, [18] discusses robustness for changing datasets while [26] addresses robustness in the context of query plan optimization. Our use-case is not so much data-driven as it is development-driven. Code changes have similar or worse impact on the performance of data management systems if not tested appropriately and continuously.

From a research perspective, the work that is closest to some of the ideas implemented in DIAMETRICS is workload compression [7] and particularly its application to index selection for relational databases [8]. This is merely part of what our framework supports and any compression algorithm can be 'plugged in' to DIAMETRICS so long as its inputs and outputs are translated to the canonical representations the various components of DIAMETRICS expect. At the same time, DIAMETRICS does not aim to provide insight into different storage configurations of a dataset to optimize its run time; rather, it provides the support necessary to compare and contrast the performance of a query engine on these configurations. Similarly, while workload characterization has received attention from the database community, it has often been used for limited-scope purposes: (*a*) as a tool to help with physical design [25]; (*b*) as a means to identify interesting queries to help in debugging SQL performance [14]; or (*c*) as a way to identify data cleaning primitives in large datasets [16].

Industry-wise, there exist commercial products that allow customers to replay entire workloads [13] in order to analyze performance [23]. The users of these products are expected to replay an entire workload, whereas we can filter it through our summarizer, in order to have something to measure the performance of an SQL engine on. Their goal is to completely replay a workload trace down to the sequence and timing of queries issued. This is not our focus: instead, we aim to provide repeatable benchmarking for a variety of systems and not the means to debug any performance issues faced by a particular deployment. Additionally, products like [13, 23] are specific to a system and lack the ability to compare and contrast multiple metrics across systems. Overall, and while certainly related to some of the components of DIAMETRICS, that line of products is less general and focuses on reactive optimization as opposed to proactive end-to-end benchmarking, which is our intention.

The effort that is most related to ours is Snowtrail [24]. While the objective is similar, i.e., testing with production data to identify performance regressions, the approach is much more limited in scope compared to DIAMETRICS. Performance regression is but one of the use-cases supported by DIAMETRICS, which is (*a*) far more general in its architecture, (*b*) provides more stand-alone components, each alleviating a particular benchmarking problem, as opposed to the monolithic design of Snowtrail, (*c*) is capable of supporting more metrics than latency, and (*d*) supports cross-system benchmarking. To the best of our knowledge, DIAMETRICS is the first system to provide a disciplined and generic end-to-end solution for benchmarking multiple query engines in a single framework.

6. CONCLUSIONS AND OUTLOOK

We presented DIAMETRICS: a framework for benchmarking query engines within Google. DIAMETRICS is a relatively new effort, that has already shown strong potential and we believe could be used in various more ways than it was originally designed for. For starters, it would be interesting to apply these techniques not only to internal customers, but also to external customers using Google's infrastracture and query engines that are interested in custom benchmarks to track the performance of Google systems on their workloads. Another interesting application of DIAMETRICS would be to use it to make configuration recommendations for new customer workloads. By measuring the similarity of a new customer's workload to existing ones we can set expectations for the performance an internal query engine will deliver. These expectations can be used to set service-level objectives for the engine itself with respect to the customer's workload. Furthermore, workload similarity may imply configuration similarity so a new customer can have a head-start with respect to optimizing a query engine's performance on their workload. Alternatively, many sample sizes of a target summarized workload can be used

to estimate the scalability of an engine for that workload, and even extrapolate to the performance of the engine as the size of the workload grows; such capability is very helpful for provisioning and planning.

Overall, DIAMETRICS solves the key problem of system benchmarking at the query engine level by providing a uniform way to develop benchmarks for multiple systems without worrying about the intricacies of each individual system. It does so in a scalable and extensible way and we believe that its modular architecture renders it as a framework that is truly greater than the sum of its parts.

7. REFERENCES

[1] D. F. Bacon, N. Bales, N. Bruno, B. F. Cooper, A. Dickinson, A. Fikes, C. Fraser, A. Gubarev, M. Joshi, E. Kogan, A. Lloyd, S. Melnik, R. Rao, D. Shue, C. Taylor, M. van der Holst, and D. Woodford. Spanner: Becoming a SQL system. In *ACM SIGMOD*, pages 331–343, 2017.

[2] D. Bitton, D. J. DeWitt, and C. Turbyfill. Benchmarking database systems: A systematic approach. In *VLDB*, pages 8–19, 1983.

[3] P. Boncz, T. Neumann, and O. Erling. Tpc-h analyzed: Hidden messages and lessons learned from an influential benchmark. In *TPCTC*, pages 61–76, 2014.

[4] M. J. Carey, D. J. DeWitt, and J. F. Naughton. The 007 benchmark. In *ACM SIGMOD*, pages 12–21, 1993.

[5] M. J. Carey, D. J. DeWitt, J. F. Naughton, M. Asgarian, P. Brown, J. E. Gehrke, and D. N. Shah. The bucky object-relational benchmark. In *ACM SIGMOD*, pages 135–146, 1997.

[6] B. Chattopadhyay, P. Dutta, W. Liu, A. Mccormick, A. Mokashi, O. Tinn, N. McKay, S. Mittal, H. ching Lee, X. Zhao, N. Mikhaylin, P. Harvey, V. Lychagina, T. Xu, B. Elliott, H. Gonzalez, L. Perez, F. Shahmohammadi, D. Lomax, and A. Zheng. Procella: A fast versatile SQL query engine powering data at YouTube. Data Works Summit, 2018.

[7] S. Chaudhuri, A. K. Gupta, and V. Narasayya. Compressing sql workloads. In *ACM SIGMOD*, pages 488–499, 2002.

[8] S. Chaudhuri and V. R. Narasayya. An efficient cost-driven index selection tool for microsoft sql server. In *VLDB*, pages 146–155, 1997.

[9] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *SoCC*, pages 143–154, 2010.

[10] A. Crolotte and A. Ghazal. Introducing Skew into the TPC-H Benchmark. In *TPCTC*, pages 137–145, 2012.

[11] S. Deep, A. Gruenheid, P. Koutris, J. Naughton, and S. Viglas. Comprehensive and efficient workload compression. *PVLDB*, 14(3):418–430, 2020.

[12] A. Dey, A. Fekete, R. Nambiar, and U. Rohm. YCSB+T: Benchmarking web-scale transactional databases. In *ICDEW*, pages 223–230, 2014.

[13] L. Galanis, S. Buranawatanachoke, R. Colle, B. Dageville, K. Dias, J. Klein, S. Papadomanolakis, L. L. Tan, V. Venkataramani, Y. Wang, and G. Wood. Oracle database replay. In *SIGMOD*, pages 1159–1170, 2008.

[14] T. Grust and J. Rittinger. Observing sql queries in their natural habitat. *ACM Trans. Database Syst.*, 38(1):3:1–3:33, Apr. 2013.

[15] A. Gupta, F. Yang, J. Govig, A. Kirsch, K. Chan, K. Lai, S. Wu, S. G. Dhoot, A. R. Kumar, A. Agiwal, S. Bhansali, M. Hong, J. Cameron, M. Siddiqi, D. Jones, J. Shute, A. Gubarev, S. Venkataraman, and D. Agrawal. Mesa: Geo-replicated, near real-time, scalable data warehousing, 2014.

[16] S. Jain and B. Howe. Data cleaning in the wild: Reusable curation idioms from a multi-year sql workload. In *QDB*, 2016.

[17] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: Interactive analysis of web-scale datasets. *PVLDB*, 3(1–2):330–339, 2010.

[18] B. Mozafari, E. Z. Y. Goh, and D. Y. Yoon. Cliffguard: A principled framework for finding robust database designs. In *SIGMOD*, pages 1167–1182, 2015.

[19] M. Pasumanskyl. Inside capacitor, bigquery's next-generation columnar storage format. In *Google Cloud Blog*, 2016.

[20] B. Samwel, J. Cieslewicz, B. Handy, J. Govig, P. Venetis, C. Yang, K. Peters, J. Shute, D. Tenedorio, H. Apte, F. Weigel, D. Wilhite, J. Yang, J. Xu, J. Li, Z. Yuan, C. Chasseur, Q. Zeng, I. Rae, A. Biyani, A. Harn, Y. Xia, A. Gubichev, A. El-Helw, O. Erling, Z. Yan, M. Yang, Y. Wei, T. Do, C. Zheng, G. Graefe, S. Sardashti, A. M. Aly, D. Agrawal, A. Gupta, and S. Venkataraman. F1 query: Declarative querying at scale. *PVLDB*, 11(12):1835–1848, 2018.

[21] J. Shute, R. Vingralek, B. Samwel, B. Handy, C. Whipkey, E. Rollins, M. Oancea, K. Littlefield, D. Menestrina, S. Ellner, J. Cieslewicz, I. Rae, T. Stancescu, and H. Apte. F1: A distributed sql database that scales. *PVLDB*, 6(11):1068–1079, 2013.

[22] Transaction Processing Performance Council. TPC Benchmark H (decision support), 2017.

[23] K. Yagoub, P. Belknap, B. Dageville, K. Dias, S. Joshi, and H. Yu. Oracle's SQL Performance Analyzer, 2008.

[24] J. Yan, Q. Jin, S. Jain, S. D. Viglas, and A. Lee. Snowtrail: Testing with production queries on a cloud database. In *DBTest*, pages 4:1–4:6, 2018.

[25] P. S. Yu, M.-S. Chen, H.-U. Heiss, and S. Lee. On workload characterization of relational database environments. *IEEE Trans. Softw. Eng.*, 18(4):347–355, Apr. 1992.

[26] J. Zhu, N. Potti, S. Saurabh, and J. M. Patel. Looking ahead makes query plans robust: Making the initial case with in-memory star schema data warehouse workloads. *PVLDB*, 10(8):889–900, 2017.

Technical Perspective of Efficient Directed Densest Subgraph Discovery

Yufei Tao
Chinese University of Hong Kong
taoyf@cse.cuhk.edu.hk

Consider a directed graph $G = (V, E)$. Given two (possibly overlapping) subsets $S, T \subseteq V$, denote by $E(S, T)$ the set of edges $(s, t) \in E$ with $s \in S$ and $t \in T$ and by $G(S, T)$ the subgraph with $S \cup T$ as the vertex set and $E(S, T)$ as the edge set. The *density* of $G(S, T)$ equals $|E(S, T)|/\sqrt{|S||T|}$. The *directed densest subgraph* (DDS) problem is to return a pair (S, T) maximizing the density of $G(S, T)$.

The problem is useful in graph mining because dense subgraphs often represent patterns deserving special attention. They could indicate, for example, an authoritative community in a social network, a building brick of more complex biology structures, or even a type of malicious behavior such as spamming. See [1, 3] and the references therein for an extensive discussion on the applications of DDS.

Previous research has led to non-trivial findings on DDS. The problem is solvable in polynomial time: the fastest algorithm known today has a time complexity $O(|V|^3|E| \log |V|)$ [2]. One could attain better efficiency by settling for an approximate output. Specifically, if the optimal density achievable is ρ^*, a pair (S, T) makes a *c-approximate solution* if $G(S, T)$ has density at least ρ^*/c. It was claimed in [2] that a 2-approximate solution could be found in $O(|V| + |E|)$ time.

The 2-approximate result of [2], unfortunately, turned out to be incorrect. Ma et al. — the authors of the paper I am introducing — pointed out a loophole in the argument of [2], which (in my opinion) was rather difficult to discern even in retrospect. They went further to disprove the claim by constructing a class of counterexamples. It remains unclear whether the issue can be fixed with the $O(|V|+|E|)$ complexity restored. Currently, the best fix available necessitates $O(|V| \cdot (|V| + |E|))$ time [3].

As a partial remedy, Ma et al. developed a 2-approximate algorithm with running time $O(\sqrt{|E|} \cdot (|V| + |E|))$, which improves $O(|V| \cdot (|V| + |E|))$ as long as $|E| = o(|V|^2)$. They achieved the purpose by establishing a connection between DDS and the $[x, y]$-*core*, a new concept that can be regarded as the directed counterpart of k-*core*. The connection then led to an elegantly simple algorithm.

Formally, call a subgraph $G(S, T)$ an $[x, y]$-*core* if every vertex in S has at least x outgoing edges and every vertex in T has at least y incoming edges. Define xy as the *product* of $G(S, T)$. Denote by $G(S^*, T^*)$ the subgraph of the maximum product. Ma et al. proved that the density of $G(S^*, T^*)$ is at least $\rho^*/2$. In other words, $G(S^*, T^*)$ serves as 2-approximate solution to the DDS problem.

Now, it remains to compute $G(S^*, T^*)$. Suppose that $G(S^*, T^*)$ is an $[x^*, y^*]$-core where the chosen values of x^* and y^* maximize x^*y^*. Assume, for the time being, $x^* \leq y^*$. Two observations are immediate. First, as $G(S^*, T^*)$ has at least x^*y^* edges[1], $x^*y^* \leq |E|$ and, hence, $x^* \leq \sqrt{|E|}$. Second, by the definition of $G(S^*, T^*)$, no $[x^*, y]$-cores can exist for any $y > y^*$.

These observations suggest the following strategy for discovering $G(S^*, T^*)$. Fix an integer $x \in [1, \sqrt{|E|}]$ and find the maximum y_x such that an $[x, y_x]$-core exists. We can accomplish this in $O(|V|+|E|)$ time in a way reminiscent of computing an undirected graph's core number:

1. $y_x = 0$
2. **repeat**
3. remove the vertices with out-degree less than x
4. y_{min} = the smallest in-degree of the remaining vertices
5. **if** $y_{min} > y_x$ **then** $y_x = y_{min}$
6. remove an arbitrary vertex with in-degree y_{min}
7. **until** no vertices are left

Besides y_x, one can also return the corresponding $[x, y_x]$-core by modifying the pseudocode slightly. Executing the code for all $x \in [1, \sqrt{|E|}]$ produces (at most) $\sqrt{|E|}$ subgraphs, among which the one with the maximum product is $G(S^*, T^*)$. The assumption $x^* \leq y^*$ can be removed by considering $y^* \leq x^*$ in a symmetric manner.

The paper of Ma et al. makes further contributions by (i) showing how to use the $[x, y]$-core technique to accelerate the state-of-the-art exact DDS algorithm [2], and (ii) demonstrating the performance of proposed algorithms through an extensive experimental evaluation. This is a beautiful paper that fuses theory and practice very nicely.

1. REFERENCES

[1] B. Bahmani, R. Kumar, and S. Vassilvitskii. Densest subgraph in streaming and mapreduce. *PVLDB*, 5(5):454–465, 2012.

[2] S. Khuller and B. Saha. On finding dense subgraphs. In *ICALP*, pages 597–608, 2009.

[3] C. Ma, Y. Fang, R. Cheng, L. V. S. Lakshmanan, W. Zhang, and X. Lin. Efficient algorithms for densest subgraph discovery on large directed graphs. In *SIGMOD*, pages 1051–1066, 2020.

[1] $|E(x^*, y^*)| \geq |S^*|x^* \geq x^*y^*$.

Efficient Directed Densest Subgraph Discovery

Chenhao Ma
Department of Computer Science, The University of Hong Kong
chma2@cs.hku.hk

Yixiang Fang
School of Data Science, The Chinese University of Hong Kong, Shenzhen
fangyixianghku@gmail.com

Reynold Cheng
Department of Computer Science, The University of Hong Kong
ckcheng@cs.hku.hk

Laks V.S. Lakshmanan
Department of Computer Science, The University of British Columbia
laks@cs.ubc.ca

Wenjie Zhang
School of Computer Science and Engineerinng, University of New South Wales
wenjie.zhang@unsw.edu.au

Xuemin Lin
School of Computer Science and Engineerinng, University of New South Wales
lxue@cse.unsw.edu.au

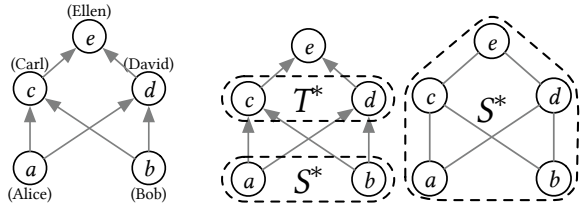
ABSTRACT

Given a directed graph G, the directed densest subgraph (DDS) problem refers to the finding of a subgraph from G, whose density is the highest among all the subgraphs of G. The DDS problem is fundamental to a wide range of applications, such as fraud detection, community mining, and graph compression. However, existing DDS solutions suffer from efficiency and scalability problems: on a three-thousand-edge graph, it takes three days for one of the best exact algorithms to complete. In this paper, we develop an efficient and scalable DDS solution. We introduce the notion of $[x, y]$-core, which is a dense subgraph for G, and show that the densest subgraph can be accurately located through the $[x, y]$-core with theoretical guarantees. Based on the $[x, y]$-core, we develop both exact and approximation algorithms. We have performed an extensive evaluation of our approaches on eight real large datasets. The results show that our proposed solutions are up to six orders of magnitude faster than the state-of-the-art.

1. INTRODUCTION

In emerging systems that manage complex relationships among objects, directed graphs are often used to model the relationships [12, 4, 1, 17]. For example, in online microblogging services (e.g., Twitter and Weibo), the "following" relationships between users can be captured as directed edges [12]. Figure 1a depicts a directed graph of the following relationship for five users in a microblogging network. Here, `Alice` has a link to `David` because she is a follower of `David`. As another example, in Wikipedia, each article can be considered as a vertex, and each link between two articles is represented by a directed edge from one vertex to another [4]. As yet another example, the Web can also be viewed as a huge directed graph [1].

In this paper, we study the problem of finding the densest subgraph from a directed graph G, which was first proposed in [13]. Conceptually, this *directed densest subgraph* (DDS) problem aims to find two sets of vertices, S^* and T^*, from



(a) A directed graph (b) Directed DS (c) Undirected DS

Figure 1: Illustrating the problem of densest subgraph discovery (or DDS problem) on the directed graph.

G, where (1) vertices in S^* have a large number of outgoing edges to those in T^*, and (2) vertices in T^* receive a large number of edges from those in S^*. To understand DDS, let us explain its usage in fake follower detection [20, 11] and community mining [15]:

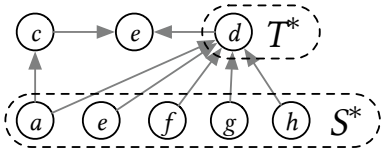


Figure 2: An example of fake follower detection.

• *Fake follower detection* [11] aims to identify fraudulent actions in social networks [11]. Figure 2 illustrates a microblogging network, with edges representing the "following" relationship. By issuing a DDS query, two sets of users S^* and T^*, are returned. Compared with other users, d (in T^*) has unusually a huge number of followers (a, e, f, g, h) in S^*. It may be worth to investigate whether d has bribed the users in S^* for following him/her.

• *Community mining* [15]. In [15], Kleinberg proposed the *hub-authority* concept for finding web communities, based on a hypothesis that a web community is often comprised of a set of *hub pages* and a set of *authority pages*. The hubs are characterized by the presence of a large number of edges to the authorities, while the authorities often receive a large number of links from the hubs. A DDS query can be issued on this network to find hubs and authorities. In Figure 3, for example, websites in S^* can be viewed as hubs providing

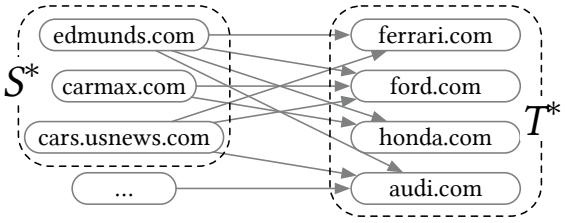


Figure 3: An example of web community.

car rankings and recommendations, while websites in T^* play the roles of authorities, as the official websites for well-known automakers.

Now let us give more details about the DDS query [13, 14, 5, 2]. Given a directed graph $G = (V, E)$ and sets of (not necesssarily disjoint) vertices $S, T \subseteq V$, the density of the directed subgraph induced by (S, T) is the number $|E(S, T)|$ of edges linking vertices in S to the vertices in T over the square root of the product of their sizes, i.e., $\rho(S, T) = \frac{|E(S,T)|}{\sqrt{|S||T|}}$. Based on this definition, the DDS problem aims to find a pair of sets of vertices, S^* and T^*, such that $\rho(S^*, T^*)$ is the maximum among all possible choices of $S, T \subseteq V$. For instance, for the directed graph in Figure 1a, the DDS is the subgraph induced by $S^*=\{a, b\}$ and $T^*=\{c, d\}$, (see Figure 1b). Its density is $\rho^* = \frac{4}{\sqrt{2\times 2}} = 2$. For each of the aforementioned applications, we can verify that the required solution corresponds to a DDS.

In undirected graphs, the density of a graph $G = (V, E)$ is defined to be $\rho(G) = \frac{|E|}{|V|}$ [9], which is different from that in directed graphs. Hence, finding the densest subgraph in undirected graphs (DS problem for short) amounts to finding the subgraph with the highest average degree [9, 7]. For example, for the undirected graph G in Figure 1c, the DS is G itself and its density is $\frac{6}{5}$, since there is no subgraph with higher density. We can observe that when $S = T$, the density of a directed graph reduces to the classical notion of the density of undirected graphs. Thus, it naturally generalizes the notion of the density of undirected graphs. On the other hand, the solution to the DDS problem returns two sets, S^* and T^*, which provide the advantage to distinguish different roles of vertices in the above applications.

Impact. The densest subgraph problem lies at the core of large scale data mining [2]. DDS is an important primitive for real-world applications, such as fake follower detection and community detection. Theoretically, the densest subgraph problem closely connects to fundamental graph problems such as network flow and bipartite matching [21]. Hence, the DDS problem receives much attention from the communities of the database, data mining, theory, and network analysis.

State-of-the-art. For a directed graph G=(V, E), we denote its number of vertices and edges by n and m respectively. In the literature, both exact [5, 14] and approximation algorithms [13, 5, 2] for DDS have been studied. The state-of-the-art exact algorithm is a flow-based algorithm [14], which mainly involves two nested loops: the outer loop enumerates all the n^2 possible values of $\frac{|S|}{|T|}$ ($1 \le |S|, |T| \le n$), while the inner loop computes the maximum density by using binary search on a flow network, regarding a specific value of $\frac{|S|}{|T|}$. The inner and outer loops take $O(nm \log n)$ and $O(n^2)$ time respectively, so its overall time complexity is $O(n^3 m \log n)$, which is prohibitively expensive for large graphs.

Table 1: Summary of exact algorithms.

Algorithm	Time complexity
`LP-Exact` [5]	$\Omega(n^6)$
`Exact` [14]	$O(n^3 m \log n)$
`DC-Exact` **(Ours)**	$O(k \cdot nm \log n)$

Note: Theoretically, $k \le n^2$. But, $k \ll n^2$, in practice.

Table 2: Summary of approximation algorithms.

Algorithm	Approx. ratio	Time complexity
`KV-Approx` [13]	$O(\log n)$	$O(s^3 n)$
`PM-Approx` [2]	$2\delta(1+\epsilon)$	$O(\frac{\log n}{\log \delta} \log_{1+\epsilon} n(n+m))$
`KS-Approx` [14]	>2	$O(n+m)$
`BS-Approx` [5]	2	$O(n^2 \cdot (n+m))$
`Core-Approx` **(Ours)**	2	$O(\sqrt{m}(n+m))$

Note: s is the sample size; ϵ, δ are the error tolerance parameters.

To improve efficiency, approximation algorithms have been developed, the most efficient one being the algorithm in [14], which only costs $O(n+m)$ time, since it iteratively peels the vertex with the smallest in-degree or out-degree. However, it was misclaimed to achieve an approximation ratio of 2, as we will show in Section 3.2. Here, the approximation ratio is defined as the ratio of the density of the DDS (i.e., the optimal solution) over that of the subgraph returned. This makes the algorithm proposed by Charikar in [5] be the best available 2-approximation algorithm, and its time complexity is $O(n^2(n+m))$. Clearly, it is still very expensive, warranting more efficient algorithms. Tables 1 and 2 summarize the properties of the exact and approximation algorithms, respectively.

Our technical contributions. To improve the state-of-the-art exact algorithm [14], we optimize its inner and outer loops. Specifically, for the inner loop, we introduce a novel dense subgraph model on directed graphs, namely [x, y]-core, inspired by the k-core [22] on undirected graphs. That is, given two sets of vertices S and T of a graph G, the subgraph induced by S and T in G is an [x, y]-core, if each vertex in S has at least x outgoing edges to vertices in T, and each vertex in T has at least y incoming edges from vertices in S. Theoretically, we show that DDS can be accurately located through the [x, y]-cores, which are often much smaller than the entire graph. As a result, we can build the flow networks on some [x, y]-cores, rather than the entire graph, which greatly improves the efficiency of computing the maximum flow. For the outer loop, we propose a divide-and-conquer strategy, which dramatically reduces the number of values of $\frac{|S|}{|T|}$ examined from n^2 to k. Theoretically, $k \in O(n^2)$. But, $k \ll n^2$, in practice. Based on the two optimization techniques above, we develop an efficient exact algorithm `DC-Exact`.

We further show that theoretically, the [x^*, y^*]-core, where x^*y^* is the maximum value among the values of x and y for all the [x, y]-cores, is a 2-approximation solution to the DDS problem. To compute the [x^*, y^*]-core, we propose an efficient algorithm, called `Core-Approx`, which completes in $O(\sqrt{m} \cdot (n+m))$ time. Therefore, compared to existing 2-approximation algorithms, it has the lowest time complexity.

We have experimentally compared our proposed solutions with the state-of-the-art solutions on eight real graphs, where the largest one consists of around two billions edges. The results show that for the exact algorithms, our proposed `DC-Exact` is over six orders of magnitude faster than the baseline algorithm on a graph with around 6,500 vertices and 51,000 edges. Besides, for approximation algorithms, our proposed `Core-Approx` algorithm can scale well to billion-scale graphs, and is also up to six orders of magnitude faster than the existing 2-approximation algorithm [5].

Outline. The rest of the paper is organized as follows. In Section 2, we formally present the DDS problem. Section 3 reviews the state-of-the-art algorithms and discusses their limitations. We present our exact and approximation algorithm in Section 4. Experimental results are presented in Section 5. We conclude this paper in Section 6. Our full version [18] provides more complete details on our algorithms and technical and empirical results.

2. PROBLEM DEFINITION

Let G=(V, E) be a directed graph, $n = |V|$ and $m = |E|$ be the number of vertices and edges in G, respectively. Given two sets $S, T \subseteq V$ which are not necessarily disjoint, we use $E(S, T)$ to denote the set of all the edges linking their vertices, i.e., $E(S, T)$=$E \cap (S \times T)$. The subgraph induced by S, T, and $E(S, T)$ is called an (S, T)-induced subgraph, denoted by $G[S, T]$. For a vertex $v \in G$, we use $d^-_G(v)$ and $d^+_G(v)$ to denote its outdegree and indegree in G respectively. Next, we formally present the density of a directed graph [13] and the problem of Directed Densest Subgraph discovery, or DDS problem. Unless mentioned otherwise, all the graphs mentioned later in this paper are directed graphs.

Definition 1 (Density of a directed graph)**.** Given a directed graph G=(V, E) and two sets of vertices $S, T \subseteq V$, the density of the (S, T)-induced subgraph $G[S, T]$ is

$$\rho(S,T) = \frac{|E(S,T)|}{\sqrt{|S| \cdot |T|}}. \quad (1)$$

The reason why the directed density chooses $\sqrt{|S||T|}$ (instead of $|S||T|$) as the denominator is that a pair of vertices with an edge between them would have the highest density and become a trivial solution if we use $|S||T|$. In Figure 1a, if we use $|S||T|$, the densities of $G[S^*, T^*]$ and $G[\{a\}, \{c\}]$ are both equal to 1. The former subgraph is denser than the latter one. [13] provides more arguments about the directed density definition.

Definition 2 (DDS)**.** Given a directed graph G=(V, E), a directed densest subgraph (DDS) D is the (S^*, T^*)-induced subgraph, whose density is the highest among all the possible (S, T)-induced subgraphs. Here, the highest density is denoted by ρ^*.

***Problem* 1** (DDS problem [13, 8, 5, 14, 2])**:** Given a directed graph G=(V, E), find a DDS[1] D=$G[S^*, T^*]$ of G.

3. EXISTING ALGORITHMS

In this section, we review the state-of-the-art exact algorithm [14] and approximation algorithms [14, 5] for the DDS problem. We remark that for approximation algorithms, both the algorithms in [14] and [5] were claimed to achieve an approximation raito of 2, but the former one runs much faster than the latter one. However, we found that the approximation ratio of the former one was misclaimed, which will be illustrated by a counter-example. Note that in this paper, the approximation ratio is defined as the ratio of the maximum density over the density of the subgraph returned.

3.1 The Exact Algorithm

The state-of-the-art exact algorithm [14] computes the DDS by solving a maximum flow problem, which generally follows the same paradigm of the exact algorithm [9] of finding the densest subgraphs on undirected graphs. We denote this algorithm by `Exact`. A flow network [10] is a directed graph F=(V_F, E_F), where there is a source node[2] s, a sink node t, and some intermediate nodes; each edge has a capacity and the amount of flow on an edge cannot exceed the capacity of the edge. The maximum flow of a flow network equals the capacity of its minimum st-cut, $\langle \mathcal{S}, \mathcal{T} \rangle$, which partitions the node set V_F into two disjoint sets, $\mathcal{S}$ and $\mathcal{T}$, such that $s \in \mathcal{S}$ and $t \in \mathcal{T}$.

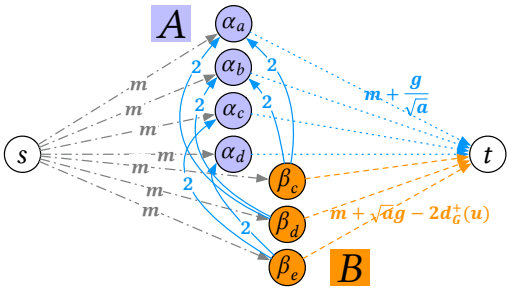


Figure 4: Illustrating the flow network.

We describe the general idea of `Exact`. Our full version [18] provides more details. It first enumerates all the possible values of a=$\frac{|S|}{|T|}$. Then, for each a, it guesses the value g of the maximum density via a binary search. After that, for each pair of a and g, it builds a flow network and runs the maximum flow algorithm to compute the minimum st-cut $\langle \mathcal{S}, \mathcal{T} \rangle$. Figure 4 depicts a flow network constructed in `Exact`. Note that if $\mathcal{S} \backslash \{s\} \neq \emptyset$, then there must be an (S, T)-induced subgraph such that its density is at least g. If such a subgraph exists and g is larger than ρ^*, we update the DDS D and its corresponding density ρ^*.

Limitations. In `Exact`, the number of possible values of a is n^2, and for each a, the while loop of binary search will have $O(\log n)$ iterations. Computing the minimum st-cut of a flow network takes $O(nm)$ time [19]. Consequently, the total time complexity of `Exact` is $O(n^3 m \log n)$, which is thus very inefficient on even small graphs.

3.2 Approximation Algorithms

The state-of-the-art approximation algorithm `KS-Approx` [14] follows the peeling paradigm. Specifically, it works in n rounds. In each round, it removes the vertex whose indegree or outdegree is the smallest, and recomputes the density of the residual graph. Finally, the subgraph whose density is the highest is returned.

[1]There might be several directed densest subgraphs of a graph, and our algorithm will find one of them.

[2]We use "node" to mean "flow network node" in this paper.

It was claimed in [14] that `KS-Approx` has an approximation ratio of 2. Unfortunately, as demonstrated by a counter-example in our full version paper [18], this claim is incorrect [3]. During our recent communication, the authors of [14] have proposed a fix which is a correct 2-approximation algorithm denoted by `FKS-Approx`, but costs $O(n \cdot (n+m))$ time.

Since `KS-Approx` is not a 2-approximation algorithm[4], the most accurate published approximation algorithm is `BS-Approx` [5], which is able to correctly find a 2-approximation result. Similar to `Exact`, `BS-Approx` enumerates all the possible values of $a=\frac{|S|}{|T|}$, and for each specific a, it iteratively removes the vertex with the minimum degree from S or T based on a predefined condition, and then updates S and T, as well as the approximate DDS $\widetilde{D}$.

Limitations. Clearly, the time complexity of `BS-Approx` is $O(n^2 \cdot (n+m))$, where the main overhead comes from the loop of enumerating all the n^2 values of a. Although it is much faster than `Exact`, it is still inefficient for large graphs. As shown in our experiments later, on a graph with about 3,000 vertices and 30,000 edges, it takes around 3 days to compute the DDS. Therefore, it is imperative to develop more efficient approximation algorithms.

4. CORE-BASED ALGORITHMS

In this section, we develop novel efficient exact and approximation algorithms for the DDS problem. Our algorithms rely on a new concept, namely $[x, y]$-core, which is an extension of the classic k-core [22] for directed graphs. In the following, we first introduce the $[x, y]$-core, then present our core-based exact algorithm, further optimize it by exploiting a divide-and-conquer strategy, and finally present our core-based 2-approximation algorithm. *Our full version [18] provides the proofs omitted in this section.*

4.1 k-core and $[x, y]$-core

We first review the definition of k-core on undirected graphs.

Definition 3 (k-core [22, 3])**.** Given an undirected graph G and an integer k ($k \geq 0$), the k-core, denoted by $\mathcal{H}_k$, is the largest subgraph of G, such that $\forall v \in \mathcal{H}_k$, $deg_{\mathcal{H}_k}(v) \geq k$.

Definition 4 ($[x, y]$-core)**.** Given a directed graph $G=(V, E)$, an (S, T)-induced subgraph $H=G[S, T]$ is called an **$[x, y]$-core**, if it satisfies:

1. $\forall u \in S, d^-_H(u) \geq x$ and $\forall v \in T, d^+_H(v) \geq y$;
2. $\nexists H' = G[S', T'] \neq H$, such that H is a subgraph of H', i.e., $S \subseteq S'$, $T \subseteq T'$, and H' satisfies (1).

We call $[x, y]$ the **core number pair** of the $[x, y]$-core, abbreviated as **cn-pair**.

Example 1. The subgraph induced by (S^*, T^*), i.e., $D = G[S^*, T^*]$ in Figure 1b is a $[2, 2]$-core. $H = G[\{a, b, c, d\}, \{c, d, e\}]$ is a $[1, 2]$-core, and D is contained in H. □

Similar to the classic k-core, the $[x, y]$-core also has some interesting properties, derived from Definition 4.

[3]The authors of [14] have confirmed that the approximation ratio of `KS-Approx` was misclaimed.

[4]We conduct an empirical study of various exact and approximation algorithms for DDS in Section 5, where we include a comparison with `FKS-Approx` and `KS-Approx`.

Lemma 1 (Nested property)**.** *An $[x, y]$-core is contained by an $[x', y']$-core, where $x \geq x' \geq 0$ and $y \geq y' \geq 0$. In other words, if $H=G[S, T]$ is an $[x, y]$-core, there must exist an $[x', y']$-core $H'=G[S', T']$, such that $S \subseteq S'$ and $T \subseteq T'$.*

Given a pair of x and y, to compute the $[x, y]$-core, we can borrow the idea of k-core decomposition [3]; that is, we can first initialize an (S, T)-induced subgraph such that $S=T=V$, then iteratively remove vertices whose indegrees (resp., outdegrees) are less than x (resp., y) from S (resp., T), and finally return the residual subgraph as the $[x, y]$-core. Clearly, computing a specific $[x, y]$-core takes $O(n+m)$ time by using the bin-sort technique in [3].

4.2 A Core-based Exact Algorithm

We first introduce an interesting lemma, then establish the relationship between the DDS and $[x, y]$-core, and finally present a core-based exact algorithm.

Lemma 2. *Given a directed graph $G=(V, E)$ and its DDS $D=G[S^*, T^*]$ with density ρ^*, we have following conclusions:*

1. for any subset U_S of S^*, removing U_S from S^* will result in the removal of at least $\frac{\rho^*}{2\sqrt{a}} \times |U_S|$ edges from D,

2. for any subset U_T of T^*, removing U_T from T^* will result in the removal of at least $\frac{\sqrt{a}\rho^*}{2} \times |U_T|$ edges from D,

where $a = \frac{|S^*|}{|T^*|}$.

Proof. We prove the lemma by contradiction. For (1), we assume that D is the DDS and removing U_S from D results in the removal of less than $\frac{\rho^*}{2\sqrt{a}} \times |U_S|$ edges from D. This implies that, after removing U_S from S^*, the density of the residual graph, denoted by $D_R=G[S^* \setminus U_S, T^*]$, will be

$$\begin{aligned}\rho(S^* \setminus U_S, T^*) = \frac{|E(S^* \setminus U_S, T^*)|}{\sqrt{|S^* \setminus U_S||T^*|}} &> \frac{\rho^*\sqrt{|S^*||T^*|} - \frac{\rho^*}{2\sqrt{a}}|U_S|}{\sqrt{(|S^*| - |U_S|)|T^*|}} \\ &= \rho^* \frac{|S^*| - \frac{|U_S|}{2}}{\sqrt{|S^*|^2 - |S^*||U_S|}} \\ &= \rho^* \frac{|S^*| - \frac{|U_S|}{2}}{\sqrt{(|S^*| - \frac{|U_S|}{2})^2 - \frac{|U_S|^2}{4}}} \\ &> \rho^*.\end{aligned}$$

However, this contradicts the assumption that D is the DDS, so the conclusion of (1) holds. Similarly, we can prove that the conclusion of (2) holds as well. Hence, the lemma holds. □

Theorem 1. *Given a graph $G=(V, E)$, the DDS $D=G[S^*, T^*]$ is contained in the $\left[\lceil\frac{\rho^*}{2\sqrt{a}}\rceil, \lceil\frac{\sqrt{a}\rho^*}{2}\rceil\right]$-core, where $a = \frac{|S^*|}{|T^*|}$.*

Since the value of ρ^* may not be known in advance, we can only locate the DDS in some cores based on $a=\frac{|S|}{|T|}$ and g guessed, by exploiting the nested property of cores. For example, given a specific a and a lower bound l of ρ^*, then we can locate the DDS in the $\left[\lceil\frac{l}{2\sqrt{a}}\rceil, \lceil\frac{\sqrt{a}l}{2}\rceil\right]$-core, since the $\left[\lceil\frac{\rho^*}{2\sqrt{a}}\rceil, \lceil\frac{\sqrt{a}\rho^*}{2}\rceil\right]$-core is nested within the $\left[\lceil\frac{l}{2\sqrt{a}}\rceil, \lceil\frac{\sqrt{a}l}{2}\rceil\right]$-core. Since the DDS is in some $[x, y]$-cores which are often

much smaller than G, we can build the flow network on these cores, rather than the entire graph G, which will significantly improve the overall efficiency.

Algorithm 1: `Core-Exact`

Input : G=(V, E)
Output: The exact DDS D=$G[S^*, T^*]$

```
1  ρ̃* ←run a 2-approximation algorithm;
2  ρ* ← ρ̃*;
3  foreach a ∈ {n1/n2 | 0 < n1, n2 <= n} do
4      l ← ρ*, r ← 2ρ̃*;
5      while r − l ≥ (√n − √(n−1)) / (n√(n−1)) do
6          g ← (l+r)/2, x ← ⌈l/(2√a)⌉, y ← ⌈√a·l/2⌉;
7          G_r ←Get-XY-Core(G, x, y);
8          F = (V_F, E_F) ← BuildFlowNetwok(G_r, a, g);
9          ⟨S, T⟩ ← Min-ST-Cut(F);
10         if S = {s} then r ← g ;
11         else
12             l ← g;
13             if g > ρ* then D ← G[S ∩ A, S ∩ B], ρ* = g;
14 return D;
```

Based on the above core-based optimization techniques, we develop a novel exact algorithm, called `Core-Exact`, which follows the same framework of `Exact`, as shown in Algorithm 1.

Analysis. To compute a specific [x, y]-core, we can complete in $O(n+m)$ time by using the idea of k-core decomposition [3]. Besides, computing the minimum st-cut takes $O(nm)$ time. Thus, the time complexity of `Core-Exact` is still $O(n^3 m \log n)$. Nevertheless, since we locate the DDS in some [x, y]-cores, the flow networks become smaller, so `Core-Exact` performs much faster than `Exact` in practice.

4.3 A Divide-and-conquer Exact Algorithm

In `Core-Exact`, we mainly optimize the inner loop of `Exact`, i.e., reducing cost of computing the minimum st-cut. A natural question comes: can we improve the outer loop of `Exact` so that we can enuemrate fewer values of a=$\frac{|S|}{|T|}$? In the following, we show that this is possible.

Our idea is based on a key observation that given a specific value of a, the results of the binary search (lines 5-13 in Algorithm 1) actually have provided insights for reducing the number of tries of a. As shown in [14], essentially the binary search solves the following optimization problem, where a is a pre-given value.

$$\max_{S,T\in V} g$$
$$\text{s.t. } \frac{|S|}{\sqrt{a}}\left(g-\frac{|E(S,T)|}{|S|/\sqrt{a}}\right)+|T|\sqrt{a}\left(g-\frac{|E(S,T)|}{|T|\sqrt{a}}\right)\le 0. \quad (2)$$

g is the maximum value the binary search can obtain when a is fixed. Then, we can derive the following lemma.

Lemma 3. *Given a graph G=(V, E) and a specific a, assume that S' and T' are the optimal choices for Equation (2). Let b=$\frac{|S'|}{|T'|}$ and c=$\frac{a^2}{b}$. Then, for any (S, T)-induced subgraph $G[S, T]$ of G, if $\min\{b,c\} \le \frac{|S|}{|T|} \le \max\{b,c\}$, we have $\rho(S,T)$≤$\rho(S',T')$.*

Algorithm 2: `DC-Exact`

Input : G=(V, E)
Output: The exact DDS D=$G[S^*, T^*]$

```
1  a_l ← 1/n, a_r ← n, ρ* ← 0, D ← ∅;
2  Divide-Conquer(a_l, a_r);
3  return D;
4  Function Divide-Conquer(a_l, a_r):
5      a_mid ← (a_l + a_r)/2;
6      run Lines 4-13 of Algorithm 1 (replace
         x ← ⌈l/(2√a)⌉, y ← ⌈√a·l/2⌉ with x ← ⌈l/(2√a_r)⌉,
         y ← ⌈√a_mid·l/2⌉);
7      let G[S', T'] be the DDS found by binary search;
8      b ← |S'|/|T'|;
9      c ← a_mid^2 / b;
10     if b > c then Swap(b, c);
11     if a_l ≤ b then Divide-Conquer(a_l, b);
12     if c ≤ a_r then Divide-Conquer(c, a_r);
```

Proof. We prove the lemma by contradiction. In Equation (2), given a specific a, let $g^*(a)$ be the optimal value of g. Because S' and T' are the optimal choices for S and T in Equation (2), S', T', and $g^*(a)$ have the following relationship, according to [14]:

$$g^*(a)=\frac{2\rho(S',T')}{\frac{\sqrt{b}}{\sqrt{a}}+\frac{\sqrt{a}}{\sqrt{b}}}. \quad (3)$$

Let $h_a(x)=\frac{\sqrt{x}}{\sqrt{a}}+\frac{\sqrt{a}}{\sqrt{x}}$. Then, we have the derivative of $h_a(x)$,

$$h'_a(x)=\frac{-a+x}{2\sqrt{a}x^{3/2}},\ x>0. \quad (4)$$

It is easy to observe that when x=a, $h'_a(x)$=0; when $x\in(0,a)$, $h'_a(x)<0$; when $x\in(a,+\infty)$, $h'_a(x)>0$. Therefore, $h_a(x)$ is a convex function, and we can get its minimum value by setting x to a.

We now prove the lemma by contradiction. Assume that there exists an [S_x, T_x]-induced subgraph, which satisfies $\min\{b,c\} \le x=\frac{|S_x|}{|T_x|} \le \max\{b,c\}$, but it has $\rho(S_x,T_y) > \rho(S',T')$. Since $h_a(x) \le h_a(b)$ and $\rho(S_x,T_y) > \rho(S',T')$, we can conclude $\frac{2\rho(S_x,T_x)}{h_a(x)} > \frac{2\rho(S',T')}{h_a(b)}$. However, this contradicts the assumption that S' and T' are the optimal choice for Equation (2). Hence, the lemma holds. □

According to Lemma 3, after conducting the binary search for a specific value of a, we can skip performing binary search for all the possible values of a in the range ($\min\{b,c\}$, $\max\{b,c\}$), so the overall efficiency can be improved dramatically. Note that since a^2=bc, we always have $a\in(\min\{b,c\}, \max\{b,c\})$.

Based on the discussions above, we develop a novel divide-and-conquer algorithm, named `DC-Exact`, as shown in Algorithm 2. First, it initialize a_l to the smallest ratio $\frac{1}{n}$, a_r to the largest ratio n, ρ^* to 0, and D to $\emptyset$ (line 1).

Complexity. The time complexity of `DC-Exact` is $O(k\cdot nm\log n)$, where k denotes the number of times invoking the binary search, which is at most n^2 since the binary search is invoked at most n^2 times in the worst case. Nevertheless, in practice we have $k \ll n^2$. As shown by our experiments later, k is often orders of magnitude smaller than n^2.

4.4 A Core-based Approximation Algorithm

While our exact algorithm, `DC-Exact`, is significantly faster than the state-of-the-art algorithm `Exact`, we can further speed it up by trading accuracy: we develop an efficient approximation algorithm, called `Core-Approx`, which achieves an approximation ratio of 2, within $O(\sqrt{m}(m+n))$ time. In the following, we first show the lower bound of density of an $[x, y]$-core.

Lemma 4 (Lower bound of density of $[x, y]$-core)**.** *Given a graph G and an $[x, y]$-core, denoted by H=$G[S, T]$, in G, the density of H is*

$$\rho(S, T) \geq \sqrt{xy}. \tag{5}$$

Next, we derive an upper bound of ρ^*. Before showing this, we introduce a novel concept called the maximum cn-pair.

Definition 5 (Maximum cn-pair)**.** Given a graph G=(V, E), a cn-pair $[x, y]$ is called the **maximum cn-pair**, if $x \cdot y$ achieves the maximum value among all the possible $[x, y]$-cores. We denote the maximum cn-pair by $[x^*, y^*]$.

Lemma 5 (Upper bound of ρ^*)**.** *Given a graph G=(V, E) and its maximum cn-pair $[x^*, y^*]$, the density ρ^* of the DDS is*

$$\rho^* \leq 2\sqrt{x^* y^*}. \tag{6}$$

Combining Lemmas 4 and 5, we obtain:

Theorem 2. *Given a graph G=(V, E), the core whose cn-pair is the maximum cn-pair, i.e., $[x^*, y^*]$-core, is a 2-approximation solution to the DDS problem.*

To compute the $[x^*, y^*]$-core, a straightforward method is to compute all the cores of G and then return the one with maximum core number pair. It is easy to observe that this method takes $O(n(n+m))$ time, because for each specific x, it costs $O(n+m)$ time to compute all the $[x, y]$-cores where y ranges from 0 to its maximum value. This, however, is costly and unnecessary because we only need to find the $[x^*, y^*]$-core. To boost the efficiency, we propose a more efficient algorithm which takes only $O(\sqrt{m}(n+m))$ time. Next, we introduce two concepts to facilitate the elaboration.

Definition 6 (Maximum equal cn-pair)**.** Given a graph G=(V, E), a cn-pair $[x, x]$ is the **maximum equal cn-pair**, if x achives the maximum value among all the possible $[x, x]$-cores. We denote the maximum equal cn-pair by $[\gamma, \gamma]$.

Lemma 6. *Given a graph G=(V, E) and its maximum equal cn-pair $[\gamma, \gamma]$, for any cn-pair $[x, y]$, we have either $x \leq \gamma$ or $y \leq \gamma$, or both of them.*

Definition 7 (Key cn-pair)**.** Given a graph G=(V, E) and its maximum equal cn-pair $[\gamma, \gamma]$, the cn-pair of an $[x, y]$-core is a **key cn-pair**, if one of the following conditions is satisfied:

1. if $x \leq \gamma$, $\nexists$ $[x, y']$-core in G, such that $y' > y$;
2. if $y \leq \gamma$, $\nexists$ $[x', y]$-core in G, such that $x' > x$.

Clearly, the maximum cn-pair is also a key cn-pair. We illustrate these concepts by Example 2.

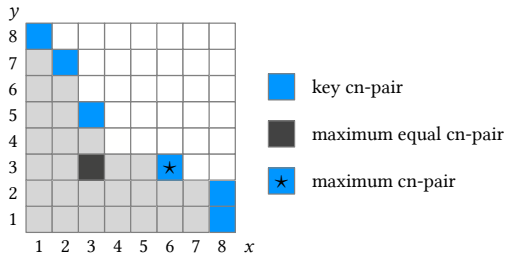


Figure 5: Illustrating the concepts of cn-pairs.

Algorithm 3: `Core-Approx`

Input : G=(V, E)
Output: An approximate DDS $\widetilde{D}$, i.e., the $[x^*, y^*]$-core

```
1  x* ← 0, y* ← 0;
2  [γ, γ] ← compute the [γ, γ]-core;
3  for x ← 1 to γ do
4      y ←GetMaxY(G, x);
5      if xy > x*y* then x* ← x, y* ← y ;
6  for y ← 1 to γ do
7      x ←GetMaxX(G, y);
8      if xy > x*y* then x* ← x, y* ← y ;
9  return compute the [x*, y*]-core;
10 Function GetMaxY(G, x):
11     S ← V, T ← V, y_max ← 0, y ← ⌊x*y*/x⌋ + 1;
12     if y > max_{u∈T}{d+_G(u)} then return y_max;
13     while |E| > 0 do
14         while ∃u ∈ T, d+_G(u) < y do
15             E ← E \ {(v,u)|v ∈ S}, T ← T \ {u};
16             while ∃v ∈ S, d−_G(v) < x do
17                 E ← E \ {(v,u)|u ∈ T}, S ← S \ {v};
18         if |E| > 0 then y_max ← y;
19         y ← y + 1;
20     return y_max;
21 Function GetMaxX(G, y):
22     reuse lines 11-20 by interchanging u with v, S with T,
       x with y, and changing y_max to x_max;
```

Example 2. Suppose that we have a graph whose cn-pairs are presented in Figure 5, where each colored cell (x, y) denotes the cn-pair of the $[x, y]$-core. Note that the blank cells do not correspond any $[x, y]$-cores. Then, the cn-pairs of the blue cells are key cn-pairs, in which the one with a star is the maximum cn-pair. The cn-pair of the black cell, i.e., [3, 3], is the maximum equal cn-pair.

Lemma 7. *Given a graph G=(V, E) and its maximum equal cn-pair $[\gamma, \gamma]$, we have $\gamma \leq \sqrt{m}$.*

By combining Lemmas 6 and 7, we get Lemma 8.

Lemma 8. *Given a graph G=(V, E), there are at most $2\sqrt{m}$ key cn-pairs in G.*

Based on the above discussions, we develop `Core-Approx`, which returns the $[x^*, y^*]$-core as an approximate DDS.

We further illustrate `Core-Approx` by Example 3.

Example 3. Reconsider the graph and its cn-pairs in Example 2. `Core-Approx` will run steps as follows: (1) finds the maximum equal cn-pair $[3, 3]$; (2) iterates x from 1 to 3 to compute the key cn-pairs whose first elements are x, i.e., $[1, 8]$, $[2, 7]$, and $[3, 5]$; (3) iterates y from 1 to 3 to

search the key cn-pairs whose second elements are y, i.e., $[8, 1]$, $[8, 2]$ and $[6, 3]$; and (4) returns the $[x^*, y^*]$-core, i.e., $[6, 3]$-core. □

Complexity. Computing the $[\gamma, \gamma]$-core takes $O(m + n)$ time as it iteratively peels vertices with the minimum indegrees or outdegrees. Similarly, functions `GetMaxY` and `GetMaxX` also complete in $O(m+n)$ time. Since there are at most $2\sqrt{m}$ key cn-pairs by Lemma 8, the total time cost of `Core-Approx` is bounded by $O(\sqrt{m}(n+m))$.

5. EXPERIMENTS

We now present the experimental results. We first discuss the setup in Section 5.1, then describe the results of exact and approximation algorithms in Sections 5.2 and 5.3.

5.1 Setup

Datasets. We use eight real datasets[5] (named MO, TC, OF, AD, AM, AR, BA, and TW in ascending order w.r.t. graph size) up to billion scale [16]. These graphs cover various domains, including social network (e.g., Twitter and Advogato), e-commerce (e.g., Amazon), and infrastructures (e.g., flight network).

Algorithms. In the experiments, we used our newly proposed exact algorithms `Core-Exact` and `DC-Exact`, and approximation algorithm `Core-Approx` to compute the DDS. Besides, we tested the following existing algorithms: `Exact` [14], `KS-Approx` [14], `FKS-Approx` [18], `PM-Approx` [2], and `BS-Approx` [5].

All the algorithms above are implemented in C++ with STL used. We run all the experiments on a machine having an Intel(R) Xeon(R) Silver 4110 CPU @ 2.10GHz processor, and 256GB memory, with Ubuntu installed.

5.2 Exact Algorithms

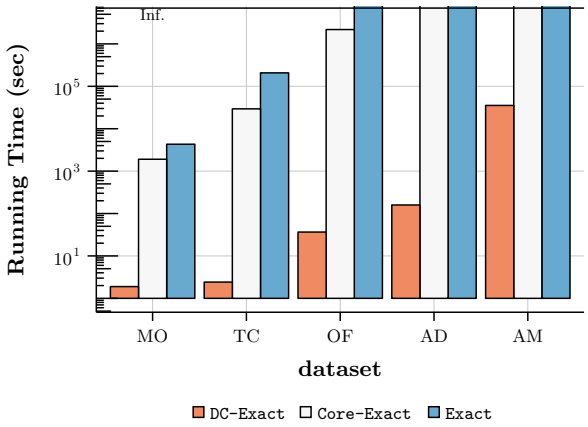


Figure 6: Efficiency of exact algorithms.

In Figure 6, we report the efficiency results of exact algorithms on the first five datasets (i.e., MO, TC, OF, AD, and AM). As these solutions cannot finish in a reasonable time on larger datasets, we do not report their results here.

[5]The datasets are available online at http://konect.uni-koblenz.de/networks/

Note that `Exact` and `Core-Exact` cannot compute the DDS within 600 hours on OF, AD, and AM.

We can observe that `Core-Exact` is at least 2× and up to 100× faster than the state-of-the-art exact algorithm `Exact`. This is mainly because `Core-Exact` locates the DDS in some $[x, y]$-cores, which are often much smaller than the entire graph, so the flow networks built on these cores become much smaller, resulting in less time cost on computing the minimum st-cut of the flow networks. Our full version [18] provides extra results about the flow network sizes during the binary search iterations.

Meanwhile, from Figure 6, we can see that `DC-Exact` is up to six and five orders of magnitude faster than `Exact` and `Core-Exact`, respectively. The main reason is that `DC-Exact` exploits a divide-and-conquer strategy, which dramatically reduces the number of $a=\frac{|S|}{|T|}$ examined. To analyze the speedup of `DC-Exact` over `Exact`, we compare the number of values of a examined in these two algorithms, which is essentially the number of times the loop of binary search is invoked. As discussed in Section 4, a is examined n^2 times in `Exact`, and k times in `DC-Exact`. Empirically, we find that k is less than 100 in the five smaller datasets. Clearly, n^2 is much larger than k. Thus, `DC-Exact` runs much faster than `Exact`.

5.3 Approximation Algorithms

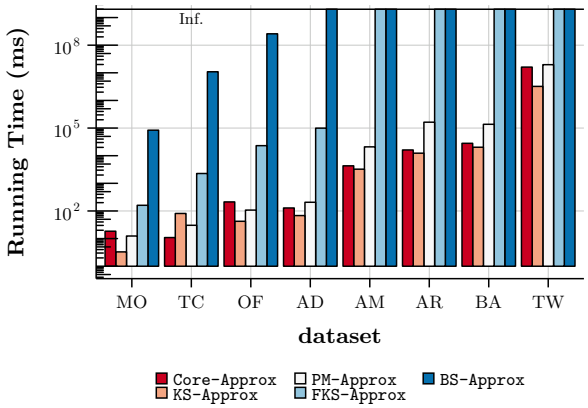


Figure 7: Efficiency of approximation algorithms.

In Figure 7, we show the running time of approximation algorithms on all the eight datasets, where bars touching the upper boundaries mean that the corresponding algorithms cannot finish within 200 hours. We can make the following observations: (1) `BS-Approx` and `FKS-Approx` are the two most inefficient algorithms, because their time complexities, i.e., $O(n^2(n+m))$ and $O(n(n+m))$, are higher than those of other algorithms. (2) `KS-Approx` is the most efficient one almost on all the datsets, since it takes only linear time cost, i.e., $O(n+m)$. However, its approximation ratio could be larger than 2, as analyzed in Section 3. (3) `Core-Approx` is the second most efficient one on almost all the datasets, followed by `PM-Approx`. (4) Among all the 2-approximation algorithms, `Core-Approx` is the fastest one, since it is up to six orders of magnitude faster than `BS-Approx`, and three orders of magnitude faster than `FKS-Approx`. Moreover, it can process billion-scale graphs. Thus, it not only obtains

high quality results, but also achieves high efficiency.

6. CONCLUSION AND FUTURE WORK

This paper studies the densest subgraph discovery on directed graphs. We show that a previous algorithm [14], which was claimed to achieve an approximation of 2, fails to satisfy the approximation guarantee. To boost the efficiency of finding DDS, we introduce a novel dense subgraph model, namely $[x, y]$-core, on directed graphs, and estalish bounds on the density of the $[x, y]$-core. We then propose a core-based exact algorithm, and further optimize it by incorporating a divide-and-conquer strategy. Besides, we find that the $[x^*, y^*]$-core, where x^*y^* is the maximum value of xy for all the $[x, y]$-cores, is a good approximation solution to the DDS problem, with theoretical guarantee. To compute the $[x^*, y^*]$-core, we develop an efficient algorithm. Extensive experiments on eight real large datasets show that both our exact and approximation algorithms are up to six orders of magnitude faster than state-of-the-art approaches.

In the future, there are several interesting research directions: (1) It would be interesting to investigate how to efficiently find the DDS with size constraints on directed graphs, such as finding a subgraph with exactly k vertices such that its density is the highest among possible subgraphs with k vertices. This problem is even more challenging than the DDS problem because of its NP-hardness. (2) In real applications, the real-world graphs often change as the time goes on, where vertices and edges may be inserted or deleted frequently. Thus, it is desirable to study how to efficiently maintain the DDS dynamically without recomputing it from scratch. (3) Recently, some researchers have developed elegant algorithms [23, 6] to decompose an undirected graph into a chain of dense subgraphs such that each one is nested within the next one and the former one has higher density than the latter one. Therefore, another exciting research direction is to perform the graph decomposition according to the directed density. (4) To the best knowledge, almost all the existing works of densest subgraph discovery model the density purely based on graph vertices and edges. However, real-world graphs are often associated with labels or attributes. As a result, it would be interesting to formulate a novel definition of graph density by considering both the links and labels or attributes, and then study how to efficiently find the densest subgraphs under this definition.

Acknowledgement

Chenhao Ma and Reynold Cheng were supported by the Research Grants Council of Hong Kong (RGC Projects HKU 17229116 and 106150091), University of Hong Kong (Projects 104005858, 104005994), the Innovation and Technology Commission of Hong Kong (ITF project MRP/029/18), and the HKU-TCL Joint Research Center for Artificial Intelligence (200009430). Lakshmanan's research was supported in part by a discovery grant and a discovery accelerator supplement grant from NSERC (Canada). Wenjie Zhang was supported by PS53783, DP200101116 and DP180103096. Xuemin Lin was supported by NSFC61232006, 2018YFB1003504, U1636215, DP200101338, DP180103096, and DP170101628.

7. REFERENCES

[1] R. Albert, H. Jeong, and A.-L. Barabási. Internet: Diameter of the world-wide web. *nature*, 401(6749):130, 1999.

[2] B. Bahmani, R. Kumar, and S. Vassilvitskii. Densest subgraph in streaming and mapreduce. *PVLDB*, 5(5):454–465, 2012.

[3] V. Batagelj and M. Zaversnik. An o(m) algorithm for cores decomposition of networks. 2003.

[4] A. Capocci, V. D. Servedio, F. Colaiori, L. S. Buriol, D. Donato, S. Leonardi, and G. Caldarelli. Preferential attachment in the growth of social networks: The internet encyclopedia wikipedia. *Physical Review E*, 74(3):036116, 2006.

[5] M. Charikar. Greedy approximation algorithms for finding dense components in a graph. In *APPROX*, pages 84–95. Springer, 2000.

[6] M. Danisch, T.-H. H. Chan, and M. Sozio. Large scale density-friendly graph decomposition via convex programming. In *WWW*, pages 233–242, 2017.

[7] Y. Fang, K. Yu, R. Cheng, L. V. Lakshmanan, and X. Lin. Efficient algorithms for densest subgraph discovery. *PVLDB*, 12(11):1719 – 1732, 2019.

[8] A. Gionis and C. E. Tsourakakis. Dense subgraph discovery: Kdd 2015 tutorial. In *KDD*, pages 2313–2314, 2015.

[9] A. V. Goldberg. *Finding a maximum density subgraph.* University of California Berkeley, CA, 1984.

[10] G. Heineman, G. Pollice, and S. Selkow. Network flow algorithms. algorithms in a nutshell, 2008.

[11] B. Hooi, H. A. Song, A. Beutel, N. Shah, K. Shin, and C. Faloutsos. Fraudar: Bounding graph fraud in the face of camouflage. In *KDD*, pages 895–904, 2016.

[12] A. Java, X. Song, T. Finin, and B. Tseng. Why we twitter: understanding microblogging usage and communities. In *WebKDD*, pages 56–65, 2007.

[13] R. Kannan and V. Vinay. *Analyzing the structure of large graphs.* University of Bonn, 1999.

[14] S. Khuller and B. Saha. On finding dense subgraphs. In *ICALP*, pages 597–608. Springer, 2009.

[15] J. M. Kleinberg. Authoritative sources in a hyperlinked environment. *JACM*, 46(5):604–632, 1999.

[16] J. Kunegis. KONECT – The Koblenz Network Collection. In *WWW*, pages 1343–1350, 2013.

[17] C. Ma, R. Cheng, L. V. Lakshmanan, T. Grubenmann, Y. Fang, and X. Li. Linc: a motif counting algorithm for uncertain graphs. *PVLDB*, 13(2):155–168, 2019.

[18] C. Ma, Y. Fang, R. Cheng, L. V. Lakshmanan, W. Zhang, and X. Lin. Efficient algorithms for densest subgraph discovery on large directed graphs. In *SIGMOD*, pages 1051–1066, 2020.

[19] J. B. Orlin. Max flows in o (nm) time, or better. In *STOC*, pages 765–774, 2013.

[20] B. A. Prakash, A. Sridharan, M. Seshadri, S. Machiraju, and C. Faloutsos. Eigenspokes: Surprising patterns and scalable community chipping in large graphs. In *PAKDD*, pages 435–448, 2010.

[21] S. Sawlani and J. Wang. Near-optimal fully dynamic densest subgraph. In *STOC*, pages 181–193, 2020.

[22] S. B. Seidman. Network structure and minimum degree. *Social networks*, 5(3):269–287, 1983.

[23] N. Tatti and A. Gionis. Density-friendly graph decomposition. In *WWW*, pages 1089–1099, 2015.

Technical Perspective: Fair Near Neighbor Search via Sampling

Qin Zhang
Indiana University
700 North Woodlawn Avenue
Bloomington, IN 47408
USA
qzhangcs@indiana.edu

One of the most important functionalities of a database system is to answer queries. We are interested in the following question: If there exists more than one answer to the given query, which one should the database report? There are two apparent choices: to return all the valid answers or to return one of them. The problem with the former choice is that it is often time-prohibitive to search for all valid answers. In the latter choice, *fairness* may become an issue, since the index built for fast search may introduce bias to the query result. For example, the index may favor a certain portion of the input data (e.g., nodes near the root of a tree index) and with a higher chance, output an answer related to that portion than other portions. Such bias can sometimes lead to undesirable consequences.

Algorithmic fairness has recently received great attention in computer science, most notably in the area of machine learning where bias in training data may be transferred to the output of the algorithm (see, e.g., [2, 1]). Conceptually, fairness can be defined as "similar items should be treated similarly", but in many settings, such as complicated social-technical systems, the precise definition of fairness is not unique and can sometimes be controversial. In the setting of database queries, fairness is often easier to define, though again the definition is not necessarily unique.

The paper "Fair Near Neighbor Search Via Sampling" by Aumüller, Har-Peled, Mahabadi, Pagh, and Silvestri studied a basic problem in similarity search called *r-near neighbor* (r-NN). In this problem, given a set of input points S, we want to build an index such that given a query point p, we can output a point q in S such that the distance between p and q is no more than r, in a computationally efficient manner with the assistance of the index. In the *fair* version of the r-NN problem, we want to return a *fair* r-near neighbor q, which is a uniform random sample from the set of all points in S within a distance of r from p.

As mentioned earlier, the idea of first enumerating all r-near neighbors and then randomly picking one is not feasible for real-time query processing. The standard technique for solving r-NN in high dimensions is *locality sensitive hashing (LSH)* [3]. In this method, we hash all points in S to a set of buckets. When a query comes, we look at points that have been hashed to the bucket that contains the query point and then try to find the answer among those points. We often repeat this process multiple times in parallel to boost the success probability of finding a r-near neighbor (if it exists). Unfortunately, the standard LSH favors close points, and the simple idea of reporting a random point from a random bucket containing the query point will not produce a "fair" solution.

The authors approach the fair r-NN problem by looking at a more generic data structure problem taking a set of sets as input. Now given a subset $\mathcal{G}$ of the input sets, we want to sample an item from $\bigcup \mathcal{G}$ uniformly at random efficiently. For the r-NN problem, $\mathcal{G}$ corresponds to the set of buckets to which the query point is hashed. One subtlety is that there may exist false positives in these buckets due to the properties of LSH. We say a point is a false positive if it is *not* a valid r-near neighbor. To handle this issue, the authors augment the generic data structure by taking into consideration a marked set of outliers $\mathcal{O}$ and updating the goal to sample uniformly at random from $\bigcup \mathcal{G} \backslash \mathcal{O}$ efficiently. This is not an easy task; the solution in the paper employs several clever technical ideas.

The paper has the potential for high impact in the emerging area of fair data structure and database system design. Despite its importance, limited research has been done in this frontier. This work can serve as a catalyst along this line of research in the years to come.

1. REFERENCES

[1] Sam Corbett-Davies, Emma Pierson, Avi Feller, Sharad Goel, and Aziz Huq. Algorithmic decision making and the cost of fairness. In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 797–806, 2017.

[2] Moritz Hardt, Eric Price, and Nati Srebro. Equality of opportunity in supervised learning. In *Advances in Neural Information Processing Systems*, pages 3315–3323, 2016.

[3] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *ACM Symposium on Theory of Computing*, pages 604–613, 1998.

Fair near neighbor search via sampling

Martin Aumüller
IT University of Copenhagen
Denmark
maau@itu.dk

Sariel Har-Peled
University of Illinois at Urbana-Champaign (UIUC)
United States
sariel@illinois.edu

Sepideh Mahabadi
Toyota Technological Institute at Chicago (TTIC)
United States
mahabadi@ttic.edu

Rasmus Pagh
BARC and University of Copenhagen
Denmark
pagh@di.ku.dk

Francesco Silvestri
University of Padova
Italy
silvestri@dei.unipd.it

ABSTRACT

Similarity search is a fundamental algorithmic primitive, widely used in many computer science disciplines. Given a set of points S and a radius parameter $r > 0$, the r-near neighbor (r-NN) problem asks for a data structure that, given any query point q, returns a point p within distance at most r from q. In this paper, we study the r-NN problem in the light of individual fairness and providing equal opportunities: all points that are within distance r from the query should have the same probability to be returned. In the low-dimensional case, this problem was first studied by Hu, Qiao, and Tao (PODS 2014). Locality sensitive hashing (LSH), the theoretically strongest approach to similarity search in high dimensions, does not provide such a fairness guarantee.

In this work, we show that LSH based algorithms can be made fair, without a significant loss in efficiency. We propose several efficient data structures for the exact and approximate variants of the fair NN problem. Our approach works more generally for sampling uniformly from a subcollection of sets of a given collection and can be used in a few other applications. The paper concludes with an experimental evaluation that highlights the inherent unfairness of NN data structures.

This is a joint version containing minor revisions of the work "Fair near neighbor search: Independent range sampling in high dimensions", published in PODS'20, June 14-19, 2020, Portland, OR, USA, ISBN 978-1-4503-7108-7/20/06, DOI: https://doi.org/10.1145/3375395.3387648 and "Near neighbor: Who is the fairest of them all?", published in the proceedings of NeurIPS'19.

1. INTRODUCTION

In recent years, following a growing concern about the fairness of the algorithms and their bias toward a specific population or feature, there has been an increasing interest in building algorithms that achieve (appropriately defined) *fairness* [13]. The goal is to remove, or at least minimize, unethical behavior such as discrimination and bias in algorithmic decision making, as nowadays, many important decisions, such as college admissions, offering home loans, or estimating the likelihood of recidivism, rely on machine learning algorithms. While algorithms are not inherently biased, nevertheless, they may amplify the already existing biases in the data.

There is no unique definition of fairness (see [17] and references therein), but different formulations that depend on the computational problem at hand, and on the ethical goals we aim for. Fairness goals are often defined in the political context of socio-technical systems, and have to be seen in an interdisciplinary spectrum covering many fields outside computer science. In particular, researchers have studied both *group fairness*–also denoted as statistical fairness–, where demographics of the population are preserved in the outcome [11], and *individual fairness*, where the goal is to treat individuals with similar conditions similarly [13]. The latter concept of "equal opportunity" requires that people who can achieve a certain advantaged outcome, such as finishing a university degree, or paying back a loan, have equal opportunity of being able to get access to it in the first place.

Bias in the data used for training machine learning algorithms is a monumental challenge in creating fair algorithms. Here, we are interested in a somewhat different problem of handling the bias introduced by the data structures used by such algorithms. Specifically, data structures may introduce bias in the data stored in them and the way they answer queries, because of the way the data is stored and how it is being accessed. It is also possible that some techniques for boosting performance, like randomization and approximation that result in non-deterministic behavior, add to the overall algorithmic bias. For instance, some database indexes for fast search might give an (unexpected) advantage to some portions of the input data. Such a defect leads to selection bias by the algorithms using such data structures. It is thus natural to want data structures that do not introduce a selection bias into the data when handling queries.

To this end, imagine a data structure that can return, as an answer to a query, an item out of a set of acceptable answers. The purpose is then to return uniformly a random item out of the set of acceptable outcomes, without explicitly computing the whole set of acceptable answers (which might be prohibitively expensive).

The Near Neighbor Problem. In this work, we study similarity search and in particular the near neighbor problem from the perspective of individual fairness. Similarity search is an important primitive in many applications in computer science such as machine learning, recommender systems, data mining, computer vision, and many others (see e.g. [5] for an overview). One of the most common formulations of similarity search is the r-near neighbor (r-NN) problem, formally defined as follows. Let $(\mathcal{X}, \mathcal{D})$ be a metric space where the distance function $\mathcal{D}(\cdot, \cdot)$ reflects the (dis)similarity between two data points. Given a set $S \subseteq \mathcal{X}$ of n points and a radius parameter r, the goal of the r-NN problem is to preprocess S and construct a data structure, such that for a query point $\mathbf{q} \in \mathcal{X}$, one can report a point $\mathbf{p} \in S$, such that $\mathcal{D}(\mathbf{p}, \mathbf{q}) \leq r$ if such a point exists. As all the existing algorithms for the *exact* variant of the problem have either space or query time that depends exponentially on the ambient dimension of $\mathcal{X}$, people have considered the approximate variant of the problem. In the *c-approximate near neighbor* (ANN) problem, the algorithm is allowed to report a point $\mathbf{p}$ whose distance to the query is at most cr if a point within distance r of the query exists, for some prespecified constant $c > 1$.

Fair Near Neighbor. As we will see, common existing data structures for similarity search have a behavior that introduces bias in the output. Our goal is to capture and algorithmically remove this bias from these data structures. Our goal is to develop a data structure for the r-near neighbor problem where we aim to be fair among "all the points" in the neighborhood, i.e., all points within distance r from the given query have the same probability to be returned. We introduce and study the *fair near neighbor* problem: if $B_S(\mathbf{q}, r)$ is the ball of input points at distance at most r from a query $\mathbf{q}$, we would like that each point in $B_S(\mathbf{q}, r)$ is returned as near neighbor of $\mathbf{q}$ with the uniform probability of $1/n(\mathbf{q}, r)$ where $n(\mathbf{q}, r) = |B_S(\mathbf{q}, r)|$.

Locality Sensitive Hashing. Perhaps the most prominent approach to get an ANN data structure is via Locality Sensitive Hashing (LSH) as proposed by Indyk and Motwani [19], which leads to sub-linear query time and sub-quadratic space. In particular, for $\mathcal{X} = \mathbb{R}^d$, by using LSH one can get a query time of $n^{\rho+o(1)}$ and space $n^{1+\rho+o(1)}$ where for the L_1 distance metric $\rho = 1/c$ [15], and for the L_2 distance metric $\rho = 1/c^2 + o_c(1)$ [5]. In the LSH framework, the idea is to hash all points using several hash functions that are chosen randomly, with the property that the collision probability between two points is a decreasing function of their distance. Therefore, closer points to a query have a higher probability of falling into a bucket being probed than far points. Thus, reporting a random point from a random bucket computed for the query, produces a distribution that is biased by the distance to the query: closer points to the query have a higher probability of being chosen. On the other hand, the uniformity property required in fair NN can be trivially achieved by finding *all* r-near neighbor of a query and then randomly selecting one of them. However, this is computationally inefficient since the query time is a function of the size of the neighborhood. One contribution in this paper is the description of much more efficient data structures that still use LSH in a black-box way.

Applications: When random nearby is better than nearest. The bias mentioned above towards nearer points is usually a good property, but is not always desirable. Indeed, consider the following scenarios:

- The nearest neighbor might not be the best if the input is noisy, and the closest point might be viewed as an unrepresentative outlier. Any point in the neighborhood might be then considered to be equivalently beneficial. This is to some extent why k-NN classification [14] is so effective in reducing the effect of noise. Furthermore, k-NN works better in many cases if k is large, but computing the k nearest neighbors is quite expensive if k is large: however, quickly computing a random nearby neighbor can significantly speed-up such classification.
- If one wants to estimate the number of items with a desired property within the neighborhood, then the easiest way to do it is via uniform random sampling from the neighborhood, for instance for density estimation [22] or discrimination discovery in existing databases [26]. This can be seen as a special case of query sampling in databases [23], where the goal is to return a random sample of the output of a given query, for efficiently providing statistics on the query.
- We are interested in anonymizing the query: returning a random near-neighbor might serve as the first line of defense in trying to make it harder to recover the query. Similarly, one might want to anonymize the nearest neighbor [24], for applications were we are interested in a "typical" data item close to the query, without identifying the nearest item.
- Popular recommender systems based on matrix factorization give recommendations by computing the inner product similarity of a user feature vector with all item feature vectors using some efficient similarity search algorithm. It is common practice to recommend those items that have the largest inner product with the user. However, in general it is not clear that it is desirable to recommend the "closest" articles. Indeed, it might be desirable to recommend articles that are on the same topic but are not *too* aligned with the user feature vector, and may provide a different perspective. As described in [1], recommendations can be made more diverse by sampling k items from a larger top-ℓ list of recommendations at random. Our data structures could replace the final near neighbor search routine employed in such systems.

To the best of our knowledge, previous results focused on exact near neighbor sampling in the Euclidean space up to three dimensions [2, 18, 23]. Although these results might be extended to $\mathbb{R}^d$ for any $d > 1$, they suffer from the *curse of dimensionality* as the query time increases exponentially with the dimension, making the data structures too expensive in high dimensions. These bounds are unlikely to be significantly improved since several conditional lower bounds show that an exponential dependency on d in query time or space is unavoidable for *exact* near neighbor search [4].

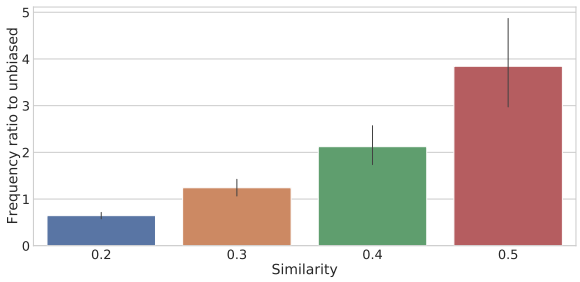


Figure 1: Bias introduced by uniform sampling from LSH buckets on the Last.FM dataset. The task is to (repeatedly) retrieve a uniform user among all users with similarity at least 0.2 to a fixed user. The result is split up into 4 buckets by rounding down the similarity to the first decimal. Error bars show the standard deviation. Compared to an unbiased sample, user vectors with small similarity are underrepresented, and users with high similarity are, by a factor of around 4 on average, overrepresented.

1.1 An example

Is a standard LSH approach really biased? As an example, we used the MinHash LSH scheme [9] to sample similar users from the Last.FM dataset used in the HetRec challenge (`http://ir.ii.uam.es/hetrec2011`). We associate each user with their top-20 artists and use Jaccard Similarity as similarity measure. We select one user at random as query, and repeatedly sample a random point from a random bucket and keep it if its similarity is above 0.2. Figure 1 reports on the ratio between the frequencies observed via this sampling approach from LSH buckets against an unbiased sample. We see a large discrepancy: the higher the similarity, the more biased the LSH is to report these points as near neighbors. This would strongly affect statistics such as estimating the average similarity of a neighbor.

1.2 Problem formulations

Here we formally define the variants of the fair NN problem that we consider in this paper. For all constructions presented in this paper, these guarantees hold only in the absence of a failure event that happens with probability at most δ for some small $\delta > 0$.

Definition 1 (r-NNIS or Fair NN). *Let $S \subseteq \mathcal{X}$ be a set of n points in a metric space $(\mathcal{X}, \mathcal{D})$. The r-*near neighbor independent sampling problem *(r-NNIS) asks to construct a data structure for S that for any sequence of up to n queries $\mathbf{q}_1, \mathbf{q}_2, \ldots, \mathbf{q}_n$ satisfies the following properties with probability at least $1 - \delta$:*

1. *For each query $\mathbf{q}_i$, it returns a point $\mathrm{OUT}_{i,\mathbf{q}_i}$ uniformly sampled from $B_S(\mathbf{q}_i, r)$;*
2. *The point returned for query $\mathbf{q}_i$, with $i > 1$, is independent of previous query results. That is, for any $\mathbf{p} \in B_S(\mathbf{q}_i, r)$ and any sequence $\mathbf{p}_1, \ldots, \mathbf{p}_{i-1}$, we have that*

$$\Pr[\mathrm{OUT}_{i,\mathbf{q}_i}{=}\mathbf{p} \mid \forall j \in [i{-}1] : \mathrm{OUT}_{j,\mathbf{q}_j}{=}\mathbf{p}_j] = \frac{1}{n(\mathbf{q}_i, r)}.$$

We also refer to this problem as Fair NN.

In the low-dimensional setting [18, 2], the r-near neighbor independent sampling problem is usually called *independent range sampling* (IRS). Next, motivated by applications, we define two approximate variants of the problem that we study in this work. More precisely, we slightly relax the fairness constraint, allowing the probabilities of reporting a neighbor to be an "almost uniform" distribution.

Definition 2 (Approximately Fair NN). *Consider a set $S \subseteq \mathcal{X}$ of n points in a metric space $(\mathcal{X}, \mathcal{D})$. The* Approximately Fair NN *problem asks to construct a data structure for S that for any query $\mathbf{q}$, returns each point $\mathbf{p} \in B_S(\mathbf{q}, r)$ with probability $\mu_\mathbf{p}$ where μ is an approximately uniform probability distribution: $\mathbb{P}(\mathbf{q}, r)/(1+\varepsilon) \leq \mu_\mathbf{p} \leq (1+\varepsilon)\mathbb{P}(\mathbf{q}, r)$, where $\mathbb{P}(\mathbf{q}, r) = 1/n(\mathbf{q}, r)$. We require the same independence guarantee as in Definition 1, i.e., the result for query $\mathbf{q}_i$ must be independent of the results for $\mathbf{q}_1, \ldots, \mathbf{q}_{i-1}$, for $i \in \{1, \ldots, n\}$.*

Second, similar to ANN, we further allow the algorithm to report an almost uniform distribution from an *approximate* neighborhood of the query.

Definition 3 (Approximately Fair ANN). *Consider a set $S \subseteq \mathcal{X}$ of n points in a metric space $(\mathcal{X}, \mathcal{D})$. The* Approximately Fair ANN *problem asks to construct a data structure for S that for any query $\mathbf{q}$, returns each point $\mathbf{p} \in S'$ with probability $\mu_\mathbf{p}$ where $\varphi/(1+\varepsilon) \leq \mu_\mathbf{p} \leq (1+\varepsilon)\varphi$, where S' is a point set such that $B_S(\mathbf{q}, r) \subseteq S' \subseteq B_S(\mathbf{q}, cr)$, and $\varphi = 1/|S'|$. We again require the same independence guarantee as in Definition 1, i.e., the result for query $\mathbf{q}_i$ must be independent of the results for $\mathbf{q}_1, \ldots, \mathbf{q}_{i-1}$, for $i \in \{1, \ldots, n\}$.*

1.3 Our results

We propose several solutions to the different variants of the Fair NN problem. Our solutions build upon the LSH data structure [15] and we denote with $\mathcal{S}(n, c)$ the space usage and with $\mathcal{Q}(n, c)$ the running time of an LSH data structure that solves the c-ANN problem in the space $(\mathcal{X}, \mathcal{D})$.

- In Section 4.2 we provide a data structure for Approximately Fair ANN that uses space $\mathcal{S}(n, c)$ and whose query time is $\widetilde{O}(\mathcal{Q}(n, c))$ *in expectation*. See Lemma 8 for the exact statement.
- Section 4.3 shows how to solve the Fair NN problem in expected query time $\widetilde{O}(\mathcal{Q}(n, c) + \frac{n(\mathbf{q}, cr)}{n(\mathbf{q}, r)})$ and space usage $O(\mathcal{S}(n, c))$. See Lemma 9 for the exact statement.

The dependence of our algorithms on ε in the approximate variant is only $O(\log(1/\varepsilon))$. While we omitted the exact poly-logarithmic factors in the list above, they are generally lower for the approximate versions. Furthermore, these methods can be embedded into existing LSH methods to achieve unbiased query results in a straightforward way. On the other hand, the exact methods will have higher logarithmic factors and use additional data structures.

A more exhaustive presentation of our results and further solutions for the Fair NN problem can be found in the full version of the paper [7]. Preliminary versions of our results were published independently in [16, 8].

1.4 Sampling from a sub-collection of sets

In order to obtain our results, we first study a more generic problem in Section 2: given a collection $\mathcal{F}$ of sets from a

universe of n elements, a query is a sub-collection $\mathcal{G} \subseteq \mathcal{F}$ of these sets and the goal is to sample (almost) uniformly from the union of the sets in this sub-collection. We also show how to modify the data structure to handle outliers in Section 3. This is useful for LSH, as the sampling algorithm needs to ignore such points once they are reported as a sample. This setup allows us to derive most of the results concerning variants of Fair NN in Section 4 as corollaries from these more abstract data structures.

Some examples of applications of a data structure that provides uniform samples from a union of sets are:

(A) Given a subset A of vertices in the graph, randomly pick (with uniform distribution) a neighbor to one of the vertices of A. This can be used in simulating disease spread [21].
(B) As shown in this work, we use variants of the data structure to implement Fair NN.
(C) Uniform sampling for range searching [18, 2]. Indeed, consider a set of points, stored in a data structure for range queries. Using the above, we can support sampling from the points reported by several queries, even if the reported answers are not disjoint.

Being unaware of any previous work on this problem, we believe this data structure is of independent interest.

2. SAMPLING FROM A UNION OF SETS

The problem. Assume you are given a data structure that contains a large collection $\mathcal{F}$ of sets of objects. In total, there are $n = |\bigcup \mathcal{F}|$ objects. The sets in $\mathcal{F}$ are not necessarily disjoint. The task is to preprocess the data structure, such that given a sub-collection $\mathcal{G} \subseteq \mathcal{F}$ of the sets, one can quickly pick uniformly at random an object from the set $\bigcup \mathcal{G} := \bigcup_{A \in \mathcal{G}} A$.

Naive solution. The naive solution is to take the sets under consideration (in $\mathcal{G}$), compute their union, and sample directly from the union set $\bigcup \mathcal{G}$. Our purpose is to do (much) better – in particular, the goal is to get a query time that depends logarithmically on the total size of all sets in $\mathcal{G}$.

Parameters. The query is a family $\mathcal{G} \subseteq \mathcal{F}$, and define $m = \mathbf{|\mathcal{G}|} := \sum_{A \in \mathcal{G}} |A|$ (which should be distinguished from $g = |\mathcal{G}|$ and from $N = |\bigcup \mathcal{G}|$).

Preprocessing. For each set $A \in \mathcal{F}$, we build a set representation such that for a given element, we can decide if the element is in A in constant time. In addition, we assume that each set is stored in a data structure that enables easy random access or uniform sampling on this set (for example, store each set in its own array).

Variants. As in Section 1.2, we consider problem variants where sample probabilities are either *exact* or *approximate*.

2.1 Almost uniform sampling

The query is a family $\mathcal{G} \subseteq \mathcal{F}$. The ***degree*** of an element $x \in \bigcup \mathcal{G}$, is the number of sets of $\mathcal{G}$ that contain it – that is, $\mathsf{d}_{\mathcal{G}}(x) = |\mathsf{D}_{\mathcal{G}}(x)|$, where $\mathsf{D}_{\mathcal{G}}(x) = \{A \in \mathcal{G} \mid x \in A\}$. We start with an algorithm (similar to the algorithm of Section 4 in [20]) that repeatedly does the following:

(I) Picks one set from $\mathcal{G}$ with probabilities proportional to their sizes. That is, a set $A \in \mathcal{G}$ is picked with probability $|A|/m$.
(II) It picks an element $x \in A$ uniformly at random.
(III) Outputs x and stops with probability $1/\mathsf{d}_{\mathcal{G}}(x)$. Otherwise, continues to the next iteration.

Since computing $\mathsf{d}_{\mathcal{G}}(x)$ exactly to be used in Step (III) is costly, our goal is instead to simulate a process that accepts x with probability *approximately* $1/\mathsf{d}_{\mathcal{G}}(x)$. We start with the process described in the following lemma.

LEMMA 1. *Assume we have g urns, and exactly $\mathsf{d} > 0$ of them, are non-empty. Furthermore, assume that we can check if a specific urn is empty in constant time. Then, there is a randomized algorithm, that outputs a number $Y \geq 0$, such that $\mathbb{E}[Y] = 1/\mathsf{d}$. The expected running time of the algorithm is $O(g/\mathsf{d})$.*

PROOF. The algorithm repeatedly probes urns (uniformly at random), until it finds a non-empty urn. Assume it found a non-empty urn in the ith probe. The algorithm outputs the value i/g and stops.

Setting $p = \mathsf{d}/g$, and let Y be the output of the algorithm. we have that

$$\mathbb{E}[Y] = \sum_{i=1}^{\infty} \frac{i}{g}(1-p)^{i-1}p = \frac{p}{g(1-p)} \cdot \frac{1-p}{p^2} = \frac{1}{pg} = \frac{1}{\mathsf{d}},$$

using the formula $\sum_{i=1}^{\infty} ix^i = x/(1-x)^2$. The expected number of probes performed by the algorithm until it finds a non-empty urn is $1/p = g/\mathsf{d}$, which implies that the expected running time of the algorithm is $O(g/\mathsf{d})$. □

The natural way to deploy Lemma 1 is to run its algorithm to get a number y, and then return 1 with probability y. The problem is that y can be strictly larger than 1, which is meaningless for probabilities. Instead, we backoff by using the value y/Δ, for some parameter Δ. If the returned value is larger than 1, we just treat it at zero. If the zeroing never happened, the algorithm would return one with probability $1/(\mathsf{d}_{\mathcal{G}}(x)\Delta)$. The probability of success is going to be slightly smaller, but fortunately, the loss can be made arbitrarily small by taking Δ to be sufficiently large.

LEMMA 2. *There are g urns, and exactly $\mathsf{d} > 0$ of them are not empty. Furthermore, assume one can check if a specific urn is empty in constant time. Let $\gamma \in (0,1)$ be a parameter. Then one can output a number $Z \geq 0$, such that $Z \in [0,1]$, and $\mathbb{E}[Z] \in I = \left[\frac{1}{\mathsf{d}\Delta} - \gamma, \frac{1}{\mathsf{d}\Delta}\right]$, where $\Delta = \lceil \ln \gamma^{-1} \rceil + 4 = \Theta(\log \gamma^{-1})$. The expected running time of the algorithm is $O(g/\mathsf{d})$. Alternatively, the algorithm can output a bit X, such that $\mathbb{P}[X = 1] \in I$.*

PROOF. We modify the algorithm of Lemma 1, so that it outputs $i/(g\Delta)$ instead of i/g. If the algorithm does not stop in the first $g\Delta + 1$ iterations, then the algorithm stops and outputs 0. Observe that the probability that the algorithm fails to stop in the first $g\Delta$ iterations, for $p = \mathsf{d}/g$, is $(1-p)^{g\Delta} \leq \exp\left(-\frac{\mathsf{d}}{g} g\Delta\right) \leq \exp(-\mathsf{d}\Delta) \leq \exp(-\Delta) \ll \gamma$.

Let Z be the random variable that is the number output by the algorithm. Arguing as in Lemma 1, we have that $\mathbb{E}[Z] \leq 1/(\mathsf{d}\Delta)$. More precisely, we have $\mathbb{E}[Z] = \frac{1}{\mathsf{d}\Delta} -$

$\sum_{i=g\Delta+1}^{\infty} \frac{i}{g\Delta}(1-p)^{i-1}p$. Let

$$\sum_{i=gj+1}^{g(j+1)} \frac{i}{g}(1-p)^{i-1}p \leq (j+1)\sum_{i=gj+1}^{g(j+1)}(1-p)^{i-1}p$$

$$= (j+1)(1-p)^{gj}\sum_{i=0}^{g-1}(1-p)^{i}p$$

$$\leq (j+1)(1-p)^{gj} \leq (j+1)\left(1-\frac{\mathsf{d}}{g}\right)^{gj} \leq (j+1)\exp(-\mathsf{d}j).$$

Let $g(j) = \frac{j+1}{\Delta}\exp(-\mathsf{d}j)$. We have that $\mathbb{E}[Z] \geq \frac{1}{\mathsf{d}\Delta} - \beta$, where $\beta = \sum_{j=\Delta}^{\infty} g(j)$. Furthermore, for $j \geq \Delta$, we have

$$\frac{g(j+1)}{g(j)} = \frac{(j+2)\exp(-\mathsf{d}(j+1))}{(j+1)\exp(-\mathsf{d}j)} \leq \frac{\left(1+\frac{1}{\Delta}\right)}{e^{\mathsf{d}}} \leq \frac{5/4}{e^{\mathsf{d}}} \leq \frac{1}{2}.$$

As such, we have that

$$\beta = \sum_{j=\Delta}^{\infty} g(j) \leq 2g(\Delta) \leq 2\frac{\Delta+1}{\Delta\exp(\mathsf{d}\Delta)} \leq 4\exp(-\Delta) \leq \gamma,$$

by the choice of value for Δ. This implies that $\mathbb{E}[Z] \geq 1/(\mathsf{d}\Delta) - \beta \geq 1/(\mathsf{d}\Delta) - \gamma$, as desired.

The alternative algorithm takes the output Z, and returns 1 with probability Z, and zero otherwise. □

LEMMA 3. *The input is a family of sets $\mathcal{F}$ that one preprocesses in linear time. Let $\mathcal{G} \subseteq \mathcal{F}$ be a sub-family and let $N = |\bigcup \mathcal{G}|$, $g = |\mathcal{G}|$, and let $\varepsilon \in (0,1)$ be a parameter. One can sample an element $x \in \bigcup \mathcal{G}$ with almost uniform probability distribution. Specifically, the probability p of an element to be output is $(1/N)/(1+\varepsilon) \leq p \leq (1+\varepsilon)(1/N)$. After linear time preprocessing, the query time is $O(g\log(g/\varepsilon))$, in expectation, and the query succeeds, with high probability (in g).*

PROOF. The algorithm repeatedly samples an element x using Steps (I) and (II). The algorithm returns x if the algorithm of Lemma 2, invoked with $\gamma = (\varepsilon/g)^{O(1)}$ returns 1. We have that $\Delta = \Theta(\log(g/\varepsilon))$. Let $\alpha = 1/(\mathsf{d}_{\mathcal{G}}(x)\Delta)$. The algorithm returns x in this iteration with probability p, where $p \in [\alpha-\gamma, \alpha]$. Observe that $\alpha \geq 1/(g\Delta)$, which implies that $\gamma \ll (\varepsilon/4)\alpha$, it follows that $(1/(\mathsf{d}_{\mathcal{G}}(x)\Delta))/(1+\varepsilon) \leq p \leq (1+\varepsilon)(1/(\mathsf{d}_{\mathcal{G}}(x)\Delta))$, as desired. The expected running time of each round is $O(g/\mathsf{d}_{\mathcal{G}}(x))$.

We prove the runtime analysis of the algorithm in the full version of the paper. In short, the above argument implies that each round, in expectation takes $O(Ng/m)$ time, where $m = \sum_{X\in\mathcal{G}}|X|$. Further, the expected number of rounds, in expectation, will be $O(\Delta m/N)$. Finally this implies that the expected running time of the algorithm is $O(g\Delta) = O(g\log(g/\varepsilon))$. □

REMARK 1. *We remark that the query time of Lemma 3 can be made to work with high probability with an additional logarithmic factor. Thus with high probability, the query time is $O(g\log(g/\varepsilon)\log N)$.*

2.2 Uniform sampling

In this section, we present a data structure that samples an element uniformly at random from $\bigcup \mathcal{G}$. The data structure uses rejection sampling as seen before but splits up all data points using random ranks. Instead of picking an element from a weighted sample of the sets, it will pick a random segment among these ranks and consider only elements whose rank is in the selected range. Let Λ be the sequence of the $n = |\bigcup \mathcal{F}|$ input elements after a random permutation; the rank of an element is its position in Λ. We first highlight the main idea of the query procedure.

Let $k \geq 1$ be a suitable value that depends on the collection $\mathcal{G}$ and assume that Λ is split into k segments Λ_i, with $i \in \{0,\ldots,k-1\}$. (We assume for simplicity that n and k are powers of two.) Each segment Λ_i contains the n/k elements in Λ with rank in $[i\cdot n/k, (i+1)\cdot n/k)$. We denote with $\lambda_{\mathcal{G},i}$ the number of elements from $\bigcup\mathcal{G}$ in Λ_i, and with $\lambda \geq \max_i\{\lambda_{\mathcal{G},i}\}$ an upper bound on the number of these elements in each segment. By the initial random permutation, we have that each segment contains at most $\lambda = \Theta((N/k)\log n)$ elements from $\bigcup\mathcal{G}$ with probability at least $1-1/n^2$. (Of course, N is *not* known at query time.)

The query algorithm works in the following steps in which all random choices are independent.

(A) Set $k = n$, and let $\lambda = \Theta(\log n)$, $\sigma_{\text{fail}} = 0$ and $\Sigma = \Theta(\log^2 n)$.
(B) Repeat the following steps until successful or $k < 2$:
  (I) Assume the input sequence Λ to be split into k segments Λ_i of size n/k, where Λ_i contains the points in $\bigcup\mathcal{F}$ with ranks in $[i\cdot n/k, (i+1)\cdot n/k)$.
  (II) Select an integer h in $\{0,\ldots,k-1\}$ uniformly at random (i.e., select a segment Λ_h);
  (III) Increment σ_{fail}. If $\sigma_{\text{fail}} = \Sigma$, then set $k = k/2$ and $\sigma_{\text{fail}} = 0$.
  (IV) Compute $\lambda_{\mathcal{G},h}$ and with probability $\lambda_{\mathcal{G},h}/\lambda$, declare success.
(C) If the previous loop ended with success, return an element uniformly sampled among the elements in $\bigcup\mathcal{G}$ in Λ_h, otherwise return $\perp$.

Since each object in $\bigcup\mathcal{G}$ has a probability of $1/(k\lambda)$ of being returned in Step (C), the result is a uniform sample of $\bigcup\mathcal{G}$. Note that the main iteration in Step (B) works for all values k, but a good choice has to depend on $\mathcal{G}$ for the following reasons. On the one hand, the segments should be small, because otherwise Step (IV) will take too long. On the other hand, they have to contain at least one element from $\bigcup\mathcal{G}$, otherwise we sample many "empty" segments in Step (II). We will see that the number k of segments should be roughly set to N to balance the trade-off. However, the number N of distinct elements in $\bigcup\mathcal{G}$ is not known. Thus, we use the naive upper bound of $k = n$. To compute $\lambda_{\mathcal{G},h}$ efficiently, we assume that, at construction time, the elements in each set in $\mathcal{F}$ are sorted by their rank.

LEMMA 4. *Let $N = |\bigcup\mathcal{G}|$, $g = |\mathcal{G}|$, $m = \sum_{X\in\mathcal{G}}|X|$, and $n = |\bigcup\mathcal{F}|$. With probability at least $1-1/n^2$, the algorithm described above returns an element $x \in \bigcup\mathcal{G}$ according to the uniform distribution. With high probability, the algorithm has a running time of $O(g\log^4 n)$.*

PROOF. We start by bounding the initial failure probability of the data structure. By a union bound, we have that the following two events hold simultaneously with probability at least $1-1/n^2$:

1. Every segment of size n/k contains no more than $\lambda = \Theta(\log n)$ elements from $\bigcup\mathcal{G}$ for all $k = 2^i$ where $i \in \{1,\ldots,\log n\}$. Since elements are initially randomly permuted, the claim holds with probability at least

$1 - 1/(2n^2)$ by suitably setting the constant in $\lambda = \Theta(\log n)$.

2. It does not happen that the algorithm reports $\perp$. The probability of this event is upper bounded by the probability p' that no element is returned in the Σ iterations where $k = 2^{\lceil \log N \rceil}$ (the actual probability is even lower, since an element can be returned in an iteration where $k > 2^{\lceil \log N \rceil}$). By suitably setting constants in $\lambda = \Theta(\log n)$ and $\Sigma = \Theta\left(\log^2 n\right)$, we get:

$$p' = \left(1 - \frac{N}{k\lambda}\right)^{\Sigma} \leq e^{-\Sigma N/(k\lambda)} \leq e^{\Theta(-\Sigma/\log n)} \leq \frac{1}{2n^2}.$$

From now on assume that these events are true.

As noted earlier, each element has a probability of $1/(k\lambda)$ of being returned, so the output are equally likely to be sampled. Note also that the guarantees are independent of the initial random permutation as soon as the two events above hold. This means that the data structure returns a uniform sample from a union-of-sets.

For the running time, first focus on the round where $k = 2^{\lceil \log N \rceil}$. In this round, we carry out $\Theta(\log^2 n)$ iterations. In Step (IV), $\lambda_{\mathcal{G},h}$ is computed by iterating through the g sets and collecting points using a range query on segment Λ_h. Since elements in each set are sorted by their rank, the range query can be carried out by searching for rank hn/k using a binary search in $O(\log n)$ time, and then enumerating all elements with rank smaller than $(h+1)n/k$. This takes time $O(\log n + o)$ for each set, where o is the output size. Since each segment contains $O(\log n)$ elements from $\bigcup \mathcal{G}$ with high probability, one iteration of Step (IV) takes time $O(g \log n)$.

The time to carry out all $\Sigma = \Theta(\log^2 n)$ iterations is thus bounded by $O(g \log^3 n)$. Observe that for all the rounds carried out before, k is only larger and thus the segments are smaller. This means that we may multiply our upper bound with $\log n$, which completes the proof. □

Using count distinct sketches to find a good choice for the number of segments k, the running time can be decreased to $O(g \log^3 n)$; we refer to the full version for more details [7].

3. HANDLING OUTLIERS

Imagine a situation where we have a marked set of outliers $\mathcal{O}$. We are interested in sampling from $\bigcup \mathcal{G} \setminus \mathcal{O}$. We assume that the total degree of the outliers in the query is at most $m_{\mathcal{O}}$ for some prespecified parameter $m_{\mathcal{O}}$. More precisely, we have $\mathsf{d}_{\mathcal{G}}(\mathcal{O}) = \sum_{x \in \mathcal{O}} \mathsf{d}_{\mathcal{G}}(x) \leq m_{\mathcal{O}}$. We get the following results by running the original algorithms from the previous section by removing outliers once we encounter them. If we encouter more than $m_{\mathcal{O}}$ outliers, we report that the number of outliers exceeds $m_{\mathcal{O}}$.

Running the algorithm described in Section 2.1 provides the guarantees summarized in the following lemma.

LEMMA 5. *The input is a family of sets $\mathcal{F}$ that one can preprocess in linear time. A query is a sub-family $\mathcal{G} \subseteq \mathcal{F}$, a set of outliers $\mathcal{O}$, a parameter $m_{\mathcal{O}}$, and a parameter $\varepsilon \in (0,1)$. One can either*

(A) Sample an element $x \in \bigcup \mathcal{G} \setminus \mathcal{O}$ with ε-approximate uniform distribution. Specifically, the probabilities of two elements to be output is the same up to a factor of $1 \pm \varepsilon$.

(B) Alternatively, report that $\mathsf{d}_{\mathcal{G}}(\mathcal{O}) > m_{\mathcal{O}}$.

The expected query time is $O(m_{\mathcal{O}} + g \log(n/\varepsilon))$, and the query succeeds with high probability, where $g = |\mathcal{G}|$, and $n = |\bigcup \mathcal{F}|$.

Running the algorithm described in Section 2.2 and keeping track of outliers has the following guarantees.

LEMMA 6. *The input is a family of sets $\mathcal{F}$ that one can preprocess in linear time. A query is a sub-family $\mathcal{G} \subseteq \mathcal{F}$, a set of outliers $\mathcal{O}$, and a parameter $m_{\mathcal{O}}$. With high probability, one can either:*

(A) Sample a uniform element $x \in \bigcup \mathcal{G} \setminus \mathcal{O}$, or

(B) Report that $\mathsf{d}_{\mathcal{G}}(\mathcal{O}) > m_{\mathcal{O}}$.

The expected query time is $O((g + m_{\mathcal{O}}) \log^4 n)$.

4. FINDING A FAIR NEAR NEIGHBOR

In this section, we employ the data structures developed in the previous sections to show the results on fair near neighbor search listed in Section 1.3.

First, let us briefly give some preliminaries on LSH. We refer the reader to [15] for further details. Throughout the section, we assume that our metric space $(\mathcal{X}, \mathcal{D})$ admits an LSH data structure.

4.1 Background on LSH

Locality Sensitive Hashing (LSH) is a common tool for solving the ANN problem and was introduced in [15].

DEFINITION 4. *A distribution $\mathcal{H}$ over maps $h\colon \mathcal{X} \to U$, for a suitable set U, is called $(r, c \cdot r, p_1, p_2)$-sensitive if the following holds for any $\mathbf{x}, \mathbf{y} \in \mathcal{X}$:*

- *if $\mathcal{D}(\mathbf{x}, \mathbf{y}) \leq r$, then $\Pr_h[h(\mathbf{x}) = h(\mathbf{y})] \geq p_1$;*
- *if $\mathcal{D}(\mathbf{x}, \mathbf{y}) > c \cdot r$, then $\Pr_h[h(\mathbf{x}) = h(\mathbf{y})] \leq p_2$.*

The distribution $\mathcal{H}$ is called an LSH family, and has quality $\rho = \rho(\mathcal{H}) = \frac{\log p_1}{\log p_2}$.

For the sake of simplicity, we assume that $p_2 \leq 1/n$: if $p_2 > 1/n$, then it suffices to create a new LSH family $\mathcal{H}_K$ obtained by concatenating $K = \Theta\left(\log_{p_2}(1/n)\right)$ i.i.d. hashing functions from $\mathcal{H}$. The new family $\mathcal{H}_K$ is (r, cr, p_1^K, p_2^K)-sensitive and ρ does not change.

The standard approach to (c, r)-ANN using LSH functions is the following. Let $\mathcal{D}$ denote the data structure constructed by LSH, and let c denote the approximation parameter of LSH. Each $\mathcal{D}$ consists of $\mathsf{L} = n^{\rho}$ hash functions $\ell_1, \ldots, \ell_{\mathsf{L}}$ randomly and uniformly selected from $\mathcal{H}$. $\mathcal{D}$ contains L hash tables $H_1, \ldots H_{\mathsf{L}}$: each hash table H_i contains the input set S and uses the hash function ℓ_i to split the point set into buckets. For each query $\mathbf{q}$, we iterate over the L hash tables: for any hash function, compute $\ell_i(\mathbf{q})$ and compute, using H_i, the set

$$H_i(\mathbf{p}) = \{\mathbf{p} : \mathbf{p} \in S, \ell_i(\mathbf{p}) = \ell_i(\mathbf{q})\} \tag{1}$$

of points in S with the same hash value; then, compute the distance $\mathcal{D}(\mathbf{q}, \mathbf{p})$ for each point $\mathbf{p} \in H_i(\mathbf{q})$. The procedure stops as soon as a (c, r)-near point is found. It stops and returns $\perp$ if there are no remaining points to check or if it found more than $3\mathsf{L}$ far points. We summarize the guarantees in the following lemma [15].

LEMMA 7. *For a given query point $\mathbf{q}$, let $S_{\mathbf{q}} = \bigcup_i H_i(\mathbf{q})$. Then for any point $\mathbf{p} \in B_S(\mathbf{q}, r)$, we have that with a probability of least $1 - 1/e - 1/3$, we have (i) $\mathbf{p} \in S_{\mathbf{q}}$ and (ii)*

$|S_{\mathbf{q}} \setminus B_S(\mathbf{q}, cr)| \leq 3\mathsf{L}$, *i.e., the number of outliers is at most* $3L$. *Moreover, the expected number of outliers in any single bucket* $S_{i,\ell_i(\mathbf{q})}$ *is at most* 1.

By repeating the construction $O(\log n)$ times, we guarantee that with high probability $B(\mathbf{q}, r) \subseteq S_{\mathbf{q}}$.

4.2 Approximately Fair ANN

For $t = O(\log n)$, let $\mathcal{D}_1, \ldots, \mathcal{D}_t$ be data structures constructed by LSH. Let $\mathcal{F}$ be the set of all buckets in all data structures, i.e., $\mathcal{F} = \left\{H_i^j(\mathbf{p}) \mid i \leq \mathsf{L}, j \leq t, \mathbf{p} \in S\right\}$. For a query point $\mathbf{q}$, consider the family $\mathcal{G}$ of all buckets containing the query, i.e., $\mathcal{G} = \{H_i^j(q) \mid i \leq L, j \leq t\}$, and thus $|\mathcal{G}| = O(\mathsf{L} \log n)$. Moreover, we let $\mathcal{O}$ to be the set of outliers, i.e., the points that are farther than cr from q. Note that as mentioned in Lemma 7, the expected number of outliers in each bucket of LSH is at most 1. Therefore, by Lemma 5, we immediately get the following result.

LEMMA 8. *Given a set* S *of* n *points and a parameter* r, *we can preprocess it such that given query* $\mathbf{q}$, *one can report a point* $\mathbf{p} \in S$ *with probability* μ_p *where* $\varphi/(1+\varepsilon) \leq \mu_{\mathbf{p}} \leq (1+\varepsilon)\varphi$, *where* S *is a point set such that* $B_S(\mathbf{q}, r) \subseteq S \subseteq B_S(\mathbf{q}, cr)$, *and* $\varphi = 1/|S|$. *The algorithm uses space* $O(\mathsf{L} \log n)$ *and its expected query time is* $O(\mathsf{L} \log n \log(n/\varepsilon))$.

4.3 Fair NN

We use the same setup as in the previous section and build $t = O(\log n)$ data structures $\mathcal{D}_1, \ldots, \mathcal{D}_t$ using LSH. We use the algorithm described in Section 2.2 with all points at distance more than r from the query marked as outliers. By the properties of the LSH and the random ranks, we expect to see $O\left(\left(\frac{n(\mathbf{q},cr)}{n(\mathbf{q},r)} + \mathsf{L}\right) \log n\right)$ points at distance at least r. This allows us to obtain the following results.

LEMMA 9. *Given a set* S *of* n *points and a parameter* r, *we can preprocess it such that given a query* $\mathbf{q}$, *one can report a point* $\mathbf{p} \in S$ *with probability* $1/n(\mathbf{q}, r)$. *The algorithm uses space* $O(\mathsf{L} \log n)$ *and has expected query time* $O\left(\left(\mathsf{L} + \frac{n(\mathbf{q},cr)}{n(\mathbf{q},r)}\right) \log^5 n\right)$.

5. EXPERIMENTAL EVALUATION

The example provided in Section 1.1 already showed the bias of sampling naively from the LSH buckets. In this section we want to consider the influence of the approximative variants discussed here, and provide a brief overview over the running time differences. A detailed experimental evaluation can be found in the full paper [7].

For concreteness, we take the MNIST dataset of handwritten digits available at `http://yann.lecun.com/exdb/mnist/`. We use the Euclidean space LSH from [12], set a distance threshold of 1250, and initialize the LSH with $\mathsf{L} = 100$ repetitions, $k = 15$, and $w = 3750$. These parameter settings provide a false negative rate of around 10%. We take 50 points as queries and test the following four different sampling strategies on the LSH buckets:

- **Uniform/Uniform**: Picks bucket uniformly at random and picks a random point in bucket.
- **Weighted/Uniform**: Picks bucket according to its size, and picks uniformly random point inside bucket.
- **Optimal**: Picks bucket according to size, and picks uniformly random point $\mathbf{p}$ inside bucket. Then it computes $\mathbf{p}$'s degree *exactly* and rejects $\mathbf{p}$ with probability $1 - 1/\deg(\mathbf{p})$.

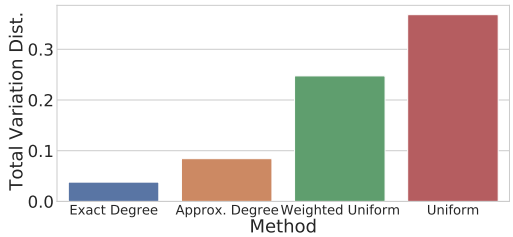


Figure 2: Total variation distance of different approaches on the MNIST dataset.

- **Degree approximation**: Picks bucket according to size, and picks uniformly random point $\mathbf{p}$ inside bucket. It approximates $\mathbf{p}$'s degree (using Lemma 1) and rejects $\mathbf{p}$ with probability $1 - 1/\deg'(p)$.

Each method removes non-close points that might be selected from the bucket. We remark that the variant Uniform/Uniform most closely resembles a standard LSH approach. Weighted/Uniform takes the different bucket sizes into account, but disregards the individual frequency of a point. Thus, the output is *not expected* to be uniform, but might be closer in distribution to the uniform distribution.

Output Distribution. For each query $\mathbf{q}$, we compute the set of near neighbors $M(\mathbf{q})$ of $\mathbf{q}$ in the LSH buckets. For each sampling strategy, we carry out the query $100|M(\mathbf{q})|$ times. The sampling results give rise to a distribution μ on $M(\mathbf{q})$, and we compare this distribution to the uniform distribution in which each point is sampled with probability $1/|M(\mathbf{q})|$. Figure 2 reports on the total variation distance between the uniform distribution and the observed distribution, i.e., $\frac{1}{2}\sum_{\mathbf{p} \in M(\mathbf{q})} |\mu(\mathbf{p}) - 1/|M(\mathbf{q})||$. As in our introductory example, we see that uniformly picking an LSH bucket results in distribution that is heavily biased. Taking the size of the buckets into account in the weighted case helps a bit, but still results in a heavily biased distribution. Even with the easiest approximation strategy for the degree, we see an improvement and achieve a total variation distance of around 0.08, with the optimal algorithm achieving around 0.04.

Differences in Running Time. With respect to running times, the approximate degree sampling provides running times that are roughly 1.5 times faster than an exact computation of the degree. The exact computation of the degree itself is around 2-3 times faster in our experiments than the most naive solution of just collecting all colliding near neighbors and selecting one at random. The methods based on rejection sampling are about a factor of 10 slower than their biased counterparts that just pick a point at random.

6. CONCLUSION AND FUTURE WORK

In this paper, we have investigated a possible definition of fairness in similarity search by connecting the notion of "equal opportunity" to independent range sampling. An interesting open question is to investigate the applicability of our data structures for problems like discrimination discovery [26], diversity in recommender systems [1], privacy preserving similarity search [25], and estimation of kernel density [10]. Moreover, it would be interesting to investi-

gate techniques for providing incentives (i.e., reverse discrimination [26]) to prevent discrimination: an idea could be to merge the data structures in this paper with distance-sensitive hashing functions in [6], which allow to implement hashing schemes where the collision probability is an (almost) arbitrary function of the distance. Finally, the techniques presented here require a manual trade-off between the performance of the LSH part and the additional running time contribution from finding the near points among the non-far points. From a user point of view, we would much rather prefer a parameterless version of our data structure that finds the best trade-off with small overhead, as discussed in [3] in another setting.

Acknowledgements. S. Har-Peled was partially supported by a NSF AF award CCF-1907400. R. Pagh is part of BARC, supported by the VILLUM Foundation grant 16582. F. Silvestri was partially supported by UniPD SID18 grant and PRIN Project n. 20174LF3T8 AHeAD.

7. REFERENCES

[1] G. Adomavicius and Y. Kwon. Optimization-based approaches for maximizing aggregate recommendation diversity. *INFORMS Journal on Computing*, 26(2):351–369, 2014.

[2] P. Afshani and J. M. Phillips. Independent range sampling, revisited again. In G. Barequet and Y. Wang, editors, *Proc. 35th Int. Symposium on Computational Geometry (SoCG)*, volume 129 of *LIPIcs*, pages 4:1–4:13, 2019.

[3] T. D. Ahle, M. Aumüller, and R. Pagh. Parameter-free locality sensitive hashing for spherical range reporting. In *Proc. 28th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, page 239–256, 2017.

[4] J. Alman and R. Williams. Probabilistic polynomials and hamming nearest neighbors. In *Proc. IEEE 56th Annual Symposium on Foundations of Computer Science (FOCS)*, page 136–150, 2015.

[5] A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Commun. ACM*, 51(1):117–122, 2008.

[6] M. Aumüller, T. Christiani, R. Pagh, and F. Silvestri. Distance-sensitive hashing. In *Proc. 37th ACM Symposium on Principles of Database Systems (PODS)*, 2018.

[7] M. Aumüller, S. Har-Peled, S. Mahabadi, R. Pagh, and F. Silvestri. Sampling a near neighbor in high dimensions — Who is the fairest of them all? *CoRR*, 2101.10905, 2021.

[8] M. Aumüller, R. Pagh, and F. Silvestri. Fair near neighbor search: Independent range sampling in high dimensions. In *Proc.39th ACM Symposium on Principles of Database Systems (PODS)*, 2020.

[9] A. Z. Broder. On the resemblance and containment of documents. In *Proc. Compression and Complexity of Sequences*, pages 21–29, 1997.

[10] M. Charikar and P. Siminelakis. Hashing-based-estimators for kernel density in high dimensions. In C. Umans, editor, *Proc. 58th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 1032–1043, 2017.

[11] A. Chouldechova. Fair prediction with disparate impact: A study of bias in recidivism prediction instruments. *Big data*, 5(2):153–163, 2017.

[12] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proc. 20th Symposium on Computational Geometry (SoCG)*, pages 253–262, 2004.

[13] C. Dwork, M. Hardt, T. Pitassi, O. Reingold, and R. Zemel. Fairness through awareness. In *Proc. 3rd Innovations in Theoretical Computer Science Conference (ITCS)*, page 214–226, 2012.

[14] B. S. Everitt, S. Landau, and M. Leese. *Cluster Analysis*. Wiley Publishing, 2009.

[15] S. Har-Peled, P. Indyk, and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. *Theory Comput.*, 8:321–350, 2012. Special issue in honor of Rajeev Motwani.

[16] S. Har-Peled and S. Mahabadi. Near neighbor: Who is the fairest of them all? In *Proc. 32th Neural Info. Proc. Sys.* (NeurIPS), pages 13176–13187, 2019.

[17] M. Hardt, E. Price, and N. Srebro. Equality of opportunity in supervised learning. In *Neural Info. Proc. Sys.* (NIPS), pages 3315–3323, 2016.

[18] X. Hu, M. Qiao, and Y. Tao. Independent range sampling. In *Proc. 33rd ACM Symposium on Principles of Database Systems (PODS)*, pages 246–255, 2014.

[19] P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proc. 30th Annu. ACM Sympos. Theory Comput.* (STOC), pages 604–613, 1998.

[20] R. M. Karp and M. Luby. Monte-carlo algorithms for enumeration and reliability problems. In *24th Symposium on Foundations of Computer Science (SFCS)*, pages 56–64. IEEE Computer Society, 1983.

[21] M. J. Keeling and K. T. Eames. Networks and epidemic models. *Journal of The Royal Society Interface*, 2(4):295–307, Sept. 2005.

[22] Y.-H. Kung, P.-S. Lin, and C.-H. Kao. An optimal k-nearest neighbor for density estimation. *Statistics & Probability Letters*, 82(10):1786 – 1791, 2012.

[23] F. Olken and D. Rotem. Sampling from spatial databases. *Statistics and Computing*, 5(1):43–57, Mar 1995.

[24] Y. Qi and M. J. Atallah. Efficient privacy-preserving k-nearest neighbor search. In *Proc. 28th International Conference on Distributed Computing Systems (ICDCS)*, page 311–319, 2008.

[25] M. S. Riazi, B. Chen, A. Shrivastava, D. S. Wallach, and F. Koushanfar. Sub-Linear Privacy-Preserving Near-Neighbor Search with Untrusted Server on Large-Scale Datasets. ArXiv:1612.01835, 2016.

[26] B. L. Thanh, S. Ruggieri, and F. Turini. k-nn as an implementation of situation testing for discrimination discovery and prevention. In C. Apté, J. Ghosh, and P. Smyth, editors, *Proc. 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 502–510, 2011.

Technical Perspective: From Sketching to Natural Language: Expressive Visual Querying for Accelerating Insight

Bill Howe
University of Washington
billhowe@uw.edu

Visualization enables effective data exploration by harnessing the higher bandwidth interactivity of the human visual cortex. But the space of possible visualizations is immense, such that general abstractions for creating (i.e., finding) the right visualization are elusive, despite recent progressin systems like vega [2] and Draco [1].

This paper provides a general abstraction, along with advanced interfaces, focusing on visualization search. If you have ever created a long sequence of visualizations looking for interesting patterns, you have manually performed a visualization search task. The visualization search problem is to find subsets of the data that, when suitably rendered, generate a visualization similar to a provided pattern specification. This task is intuitively difficult, requiring at least a model of visualization similarity, a representation of a massive search space, a strategy for navigating the search space, and appropriate interfaces through which users can express specifications. The authors approach these challenges by designing a shape query algebra consisting of primitives and operators that can describe complex functions; i.e. "trendlines." The algebra assembles complex patterns from simple segments, akin to the way complex continuous functions are approximated by assemblies of piecewise linear functions in engineering and science (e.g., finite element models).

The key insight of the ShapeSearch algebra is that reasoning about regions of a function that are rising, falling, or flat is simple enough to capture a user's notion of pattern, but expressiv enough to model complex patterns. For example, a peak pattern rises then falls, a plateau rises and is then flat. By including expressions that can reference preceding or subsequent segments, the language can capture patterns such as "rising, then rising more sharply." In addition to sequencing via concatenation, conjunctions and disjunctions are supported; e.g., the value should rise AND not fall in a given region. Other convenience extensions include quantifiers (the value should rise at least once and at most twice), nesting (a peak within a given range), and iteration (look for a peak anywhere within a given range); these extensions can be rewritten using core primitives.

Shapesearch expressions are designed to be ambiguous — matching and scoring ambiguous expressions against dataset is another key contribution. Each segment is scored using an inverse tangent expression to produce a value between -1 and 1; these segment scores are then aggregated to produce expression scores. Fuzzy matches involve underspecified queries, where the start and/or end point of a region is left to the system to find — a potentially expensive search. A dynamic programming algorithm provides an accurate but expensive baseline, while a provably optimal approximate algorithm significantly reduces the search space by observing that start and end points will typically fall at critical points of the underlying function (i.e., where the slope changes direction), then matching and merging the query segments, greedily, according to their score.

Shapesearch supports a natural language interface for novice users, speech-to-text scenarios, and integration with other NL services, a regular expression interface for more complex expressions, and a sketching interface for drawing patterns by hand. The sketching interface promotes visual patterns as the primary model for reasoning about subsets of data — not equations, not complicated queries or code, not unreliable labels or metadata. It is this idea that separates ShapeSearch from other proposals for visualization search, including Draco [1] and Voyager [3].

Shapesearch represents a significant departure from conventional data analysis approaches by lifting up approximate visual patterns as a first class component of interaction and computation. This work has the potential to help develop a class of software and methods subscribing to this view, and, as the authors point out, immediately suggests some intriguing directions for future work: better interfaces to support mixing sketches and code, auto-complete features to support incremental composition of complex queries, and search over different attributes simultaneously.

1. REFERENCES

[1] D. Moritz, C. Wang, G. Nelson, H. Lin, A. M. Smith, B. Howe, and J. Heer. Formalizing visualization design knowledge as constraints: Actionable and extensible models in draco. *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)*, 2019.

[2] A. Satyanarayan, D. Moritz, K. Wongsuphasawat, and J. Heer. Vega-lite: A grammar of interactive graphics. *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)*, 2017.

[3] K. Wongsuphasawat, Z. Qu, D. Moritz, R. Chang, F. Ouk, A. Anand, J. Mackinlay, B. Howe, and J. Heer. Voyager 2: Augmenting visual analysis with partial view specifications. In *ACM Human Factors in Computing Systems (CHI)*, 2017.

From Sketching to Natural Language: Expressive Visual Querying for Accelerating Insight

Tarique Siddiqui
Microsoft Research
tasidd@microsoft.com

Paul Luh
University of Illinois (UIUC)
luh2@illinois.edu

Zesheng Wang
University of Illinois (UIUC)
zwang180@illinois.edu

Karrie Karahalios
University of Illinois (UIUC)
kkarahal@illinois.edu

Aditya G. Parameswaran
UC Berkeley
adityagp@berkeley.edu

ABSTRACT

Data visualization is the primary means by which data analysts explore patterns, trends, and insights in their data. Unfortunately, existing visual analytics tools offer limited *expressiveness* and *scalability* when it comes to searching for visualizations over large datasets, making visual data exploration labor-intensive and time-consuming. We first discuss our prior work on Zenvisage that helps accelerate exploratory data analysis via an interactive interface and an expressive visualization query language, but offers limited *flexibility* when the pattern of interest is under-specified and approximate. Motivated from our findings from Zenvisage, we develop ShapeSearch, an efficient and flexible pattern-searching tool that enables the search for desired patterns via multiple mechanisms: sketch, natural-language, and visual regular expressions. ShapeSearch leverages a novel *shape querying algebra* that can express a large class of shape queries and supports query-aware and perceptually-aware optimizations to execute shape queries within interactive response times. To further improve the usability and performance of both Zenvisage and ShapeSearch, we discuss a number of open research problems.

1. INTRODUCTION

With the pressing need to derive value from data, domain experts, spanning virtually all sectors of society, spend considerable time exploring data to identify patterns and trends. These domain experts often have limited understanding of programming and hence rely heavily on visual analytic tools such as Excel and Tableau to understand their data. The state of the art for domain experts is to load their data into a visualization tool, and repeatedly generate visualizations until the desired patterns or insights are identified. Unfortunately, this repeated process of manual examination to scour for desired insights becomes painful, tedious, and time-consuming as the size and complexity of datasets increase. Even on moderately sized datasets, a domain scientist may need to examine as many as tens of thousands of visualizations, all to test a single hypothesis, a severe impediment to data exploration.

We characterize this problem of *visualization search* using examples from genomic data analysis.

Motivating Example. *Genomic researchers often study genes during clinical trials, e.g., how genes affect clinical trial outcomes, how the behavior of genes get affected on specific medications, etc. As an example, given a dataset consisting of clinical trial outcomes (positive vs. negative), researchers often want to find genes that can visually explain the differences in these outcomes. To do so, current tools require the researchers to manually generate tens of thousands of scatter plots—with the x- and y- axes each referring to a gene, and each outcome depicted as a point in the scatterplot—to determine whether the outcomes can be clearly distinguished in the scatter plot.*

Similarly, researchers study changes in gene expressions while investigating the impact of drugs on disease treatment. For doing so, they often explore trend line visualizations, one corresponding to each gene, with the x-axis as days, and the y-axis as the expression values. For example, when influenced by an external factor, a gene can get induced (up-regulated), or repressed (down-regulated), or can have both induced or repressed pattern within a certain time window. Based on their domain understanding, researchers first hypothesize the expected change in expression that an affected gene should depict. They, then, generate thousands of visualizations, one for each gene, and manually inspect them for the hypothesized patterns.

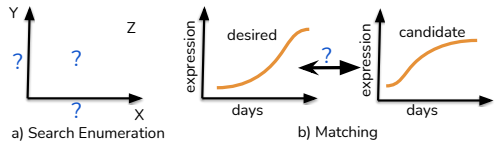


Figure 1: Two Facets of the Visualization Search Problem

In both of the above scenarios, the common theme is the manual examination of a large number of generated visualizations for a specific visual pattern. As depicted in Figure 1, there are two facets to this visualization search problem. First, it is challenging for users to specify the search space of visualizations they are interested in, which forces them to manually generate a large collection of visualizations. The space of visualizations is determined by the number of possible attributes for X and Y axes, aggregation functions and possible subsets of data (denoted by symbol Z in Figure 1a). This space grows exponentially as the size and number of attributes in the data increases. The second facet deals with *visualization matching*. Given a specific pattern of interest, users are typically interested in a subset of visualizations that closely match this pattern. Unfortunately, existing visualization tools are not *expressive* enough to capture either of the two facets.

Our first attempt to address these challenges resulted in a visual data exploration system, Zenvisage [11,23,24]. Zenvisage takes as input a high level specification of what the user wants and automatically identifies the relevant visualizations. It supports an interactive interface that allows users to quickly search for simple patterns

via sketching. For expressing more complex search enumeration and visualization matching, Zenvisage supports ZQL—an expressive visualization exploration language that lets users operate over a collection of visualizations using a core set of primitives (e.g., comparison, filtering, sorting) based on visual patterns.

While Zenvisage is an useful first step in solving the visualization search problem, it only supports standard distance measures such as Euclidean distance for matching visualizations, thereby lacking *flexibility* in terms of how a visualization is matched. For instance, it is unable to support search when the desired shape is under-specified or approximate, e.g., finding products whose sales is decreasing over some 3 month window, without specifying when, or those whose sales has many increasing and decreasing portions, without specifying when these portions occur, their magnitude or their width.

Thus, to support such flexible querying mechanisms, we developed ShapeSearch [25, 26], a pattern querying system that supports multiple mechanisms for helping users express and search for desired visual patterns. ShapeSearch incorporates an expressive shape query algebra consisting of shape-based primitives and operators, for expressing a large variety of patterns in trendlines. We developed this algebra after discussions with domain experts, including those from astronomy and genomics, as well as studying a large corpus of pattern queries collected via Mechanical Turk.

ShapeSearch supports multiple specification mechanisms that are internally translated to a shape query algebra representation: ShapeSearch supports a *natural language interface*, coupled with a sophisticated parser and translator for translating them into the algebra. ShapeSearch also supports a *sketching interface* for simpler patterns, and returns visualizations that precisely match the drawn trends. To support more complex needs, the system provides a *visual regular expression* language for issuing queries that cannot be easily expressed via natural language or sketching. The three interfaces can be used simultaneously and interchangeably, as user needs and pattern complexities evolve.

Finally, for ensuring interactive response times on ad-hoc queries, ShapeSearch leverages a pattern-matching engine that relies on minimal pre-processing or indexing. Directly generating and processing a large collection of visualizations, where each visualization has thousands of values, can lead to a long response time. Instead, ShapeSearch uses *perceptually-aware* pattern scoring mechanisms and *query-aware* optimizations—that help prune a large number of visualizations and/or parts of visualizations, for effective and efficient pattern matching.

Outline. The rest of our paper is organized as follows. We first discuss our experiences from our prior work on Zenvisage that motivated us to develop ShapeSearch, describing a simple interactive interface and ZQL (Section 2). We then give an overview of ShapeSearch, discussing how it addresses the limitations of Zenvisage (Section 3). Next, we dive into the details of shape algebra that makes the core of ShapeSearch (Section 4). We then describe efficient algorithms for executing shape queries (Section 5). We discuss how we support natural language queries in ShapeSearch (Section 6). In the end, we discuss future directions to further improve the usability and performance of both Zenvisage and ShapeSearch (Section 7).

2. EXPERIENCES FROM ZENVISAGE

Zenvisage is a visual analytics system that supports an interactive interface for searching for visualization with simple patterns, along with an expressive query language for more complex queries. We briefly discuss each of these modes and then describe the findings from our user evaluation.

2.1 Interactive Search Interface

Figure 2 shows the interactive search interface of Zenvisage loaded with a real estate dataset.

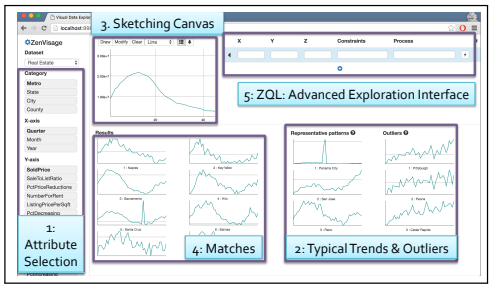


Figure 2: Zenvisage *interactive visual query interface*

Attribute Selection. The first step is attribute selection (Box 1). Here the user can specify the desired X axis attribute, and the desired Y axis attribute for the visualization(s) that the user is interested in. In this case, the user has specified the X axis as quarters (in other words, time), and the Y axis as the real-estate sold price. Additionally, the user specifies the category: this is a variable indexing the space of candidate visualizations the user is operating over. Here, the selected category is "metro"—indicating a metro area or township. We depicted the category as "Z" in Figure 1a.

Summarization of Typical and Outlier Trends. As soon as the user selects the X, Y and category, immediately, Zenvisage populates Box 2 with typical or representative trends across categories, and outliers. In this case, there are three typical trends that were found across different metros (i.e., categories): one corresponding to a spike in the middle (Panama City), one to a gradual increasing trend (San Jose), and one to a trend that increased and then decreased (Reno)—most of the other trends were found to be similar to one of these three. The outlier visualizations (Pittsburgh, Peoria, Cedar Rapids) have a large number of seemingly random spikes.

Drawing or Drag-and-Drop Canvas. Then, in Box 3, the editable canvas, the user can either draw a shape that they are looking for, or alternatively drag and drop one of the displayed visualizations into the canvas. In this manner, the user indicates that they would like to see a similarity search starting from the shape or pattern that they have drawn or dragged onto the canvas. The user is also free to edit the drawn pattern. In this figure, the user has drawn a trend which is gradually increasing up, then gradually decreasing after that.

Similarity Search Results. As soon as the user completes an interaction in Box 3, Box 4 is populated with results corresponding to visualizations (on varying the category) that are most similar to the trend in Box 3, ordered by similarity. The system allows users to choose between three different similarity metrics. Currently, the three metrics Zenvisage provides are Euclidean Distance, DTW, and Segmentation [24].

Overall, this interactive search interface satisfies simple pattern search needs via sketching and drag-and drop, and provides context via representative and outlier patterns. However, it offers limited expressiveness when it comes to more complex data exploration needs. For instance, it is difficult to search for visualizations across a wide range of X and Y attributes (recall that before sketching, we need to set the X and Y axis to specific attributes), or compare two visualizations without using the drawing canvas (e.g., finding 2 products that similar revenue and profit trends over years). Furthermore, one cannot specify multi-step queries involving search for multiple patterns simultaneously, e.g., find products with increasing sales trend in Europe but decreasing sales trend in the US. For supporting these more complex needs, we introduced a second mode, called ZQL, short for Zenvisage Query Language, that users can specify in Box 5 in Figure 4.

2.2 ZQL: A Visualization Querying Language

ZQL is a high level language that automates the manual visual data exploration process by allowing users to specify their desired

Name	X	Y	Z	Process
f1	year	soldprice	z1 <– 'state'.'*'	z2 <– $argmax_{z1}[k=1]D(f1,f2)$
f2	year	soldpricepersqft	z1	
*f3	year	{soldprice, soldpricepersqft}	z2	

Table 1: A ZQL *query retrieving visualizations for a state where the soldprice over year trends are most dissimilar to the soldpricepersqft trend.*

Name	X	Y	Z	Process
f1	x1 <– *	y1 <– *	'state'.'CA'	x2,y2 <– $argmax_{x1,y1}[k=1]D(f1,f2)$
f2	x1	y1	'state'.'NY'	
*f3	x2	y2	'state'.{'NY', 'CA'}	

Table 2: A ZQL *query retrieving two different visualizations (among different combinations of x and y) for states of CA and NY that are the most dissimilar.*

visualization objective in a few lines. Instead of providing the low-level data retrieval and manipulation operations, users operate at the level of *sets of visualizations, and compare, sort, filter, and transform* visualizations as well as attributes—eventually visualized on either the X or Y axis, or used to sub-select the set of data that is visualized.

We describe the capabilities of ShapeQuery via two examples (depicted via Table 1 and Table 2). Consider the first example where we want to find the states where the soldprice trend is most similar to the soldpricepersqft (i.e., sold price per square foot) trend. Table 1 depicts a 3-line ZQL query for this task. We first compose two collections of visualizations. The first row composes the first collection with X = year, Y = avg(soldprice), and Z = state.*, consisting of one visualization for each possible state. The Z column corresponds to the Category header in the previous section, indicating the space of visualizations over which the user is operating—in this case, the Z column is fairly simple, there is a single visualization, corresponding to each state. Similarly, the second row composes the second collection with X and Z column stay similar and Y is set to avg(soldpricepersqft).

Once we have composed the two visualization collections (referred via f1 and f2),the Process column is used to compare, sort, and filter the visualizations between the collections. In this example, we iterate the visualizations for each state (notice the variable z1) in f1 and f2 and compare them using a functional primitive D, computing distance, via D(f1, f2). Then, argmin is a sort-filter primitive that sorts the states based on distance scores and selects the top 1 state with minimum scores. Finally, in row 3, we output the overall sales over year visualizations for the selected products as bar-charts. The * in *f3 indicates that these visualizations are to be output to the user.

As another example, say we are interested in finding a pair of X and Y axes where the visualizations for two specific states 'NY' and 'CA' differ the most. For doing this, we write a ZQL query depicted in Table 2. In the first line, we fetch all visualizations for the states 'NY' that can be formed by having different combinations of X and Y axes. Similarly in the second row, we retrieve all possible visualizations for the product 'stapler'. In the process column, we iterate over the possible pairs of X and Y axes values, compare the corresponding visualizations in f1 and f2 and finally select the pair of X and Y axis values where the two products differ the most. In the last two rows, we output these visualizations.

Overall, ZQL can capture a wide range of visual exploration queries, including drill-downs and filtering based on specific patterns. We formally describe the expressive power of ZQL using a visual exploration algebra in [23].

2.3 Takeaways from User Evaluation

To understand the utility of Zenvisage, we conducted user studies with both novice and experienced data analysts [23], as well as case-studies with collaborating researchers from domains such as genomics, astronomy, and battery science [11].

Our findings show that Zenvisage enables faster and more accurate exploration compared to existing visualization tools such as Tableau, which require considerable manual exploration for finding visualizations with specific patterns. Users who had worked with MATLAB, Python, and R said that ZQL can lead to faster initial exploration of data without requiring to write a lot of code. Those having experience with SQL found ZQL a lot less complicated, less verbose and faster when it comes to comparing subsets of data [23]. Similarly, our collaborating researchers have used Zenvisage for various findings, including the fact that a dip in a light curve was caused by malfunctioning equipment (for astronomy), the fact that a relationship between two specific physical properties of electrolytes was independent of a third one (for battery science), and for reproducing of characteristic gene expression profiles from a recent paper (for genetics) [11].

While Zenvisage offers a promising first step to the problem of painful manual exploration of visualizations, the underlying challenge of visualization search is far from solved. We discovered two main challenges. One pertains to the usability of ZQL. In order to leverage ZQL, domain experts need to learn and switch to a new querying language, a major hindrance to its broader adoption. Domain experts with prior experience with computational notebooks often expressed a need for transitioning between writing code and using ZQL abstractions. Additionally, instead of writing their queries in one step, users often intended to construct them in an incremental manner using prior queries as context. In Section 7, we discuss these issues and potential solutions in more detail, highlighting another system LUX [1] from Lee et al. that partially addresses these issues.

The second challenge with Zenvisage deals with how visualizations are matched. For the rest of the paper, we focus on this challenge and present a new system ShapeSearch to address it.

2.3.1 The Problem of Flexible Shape Matching

The sketch-based interface in Zenvisage as well as other similar visualization searching tools [7, 13, 27] offer limited flexibility in terms of how a visualization is matched. For instance, visualization search often involves pattern matching where the desired pattern of interest is under-specified and approximate, e.g., finding stocks whose prices are decreasing for some time, followed by a sharp rise, with the position and intensity of movements being left unspecified, or when the desired shape is complex, e.g., finding gene expression profiles where there is an unspecified number of peaks and valleys followed by a flattening out. We highlight the key characteristics of such pattern matching tasks below.

Fuzzy Matching. Domain experts (i) typically search for patterns that are *approximate*, and are often not interested in the specific details or local fluctuations as much as the overall shape, and (ii) they often *do not* specify or even know the exact location of the occurrence of patterns. For example, biologists routinely look for structural changes in gene expression, e.g., rising and falling at different times (Figure 3a), characterizing internal biological processes such as the cell cycle or circadian rhythms, or external perturbation, such as the influence of a drug or presence of a disease.

Searching Multiple Simple Patterns. We notice that domain experts often describe complex patterns using a *combination of multiple simple ones*. Each individual pattern is typically described

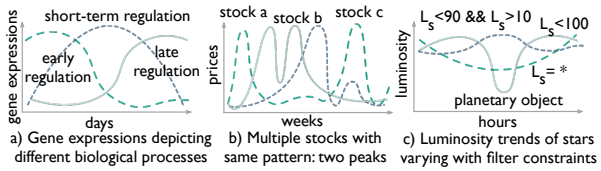


Figure 3: Shapes characterizing real world phenomena

using words such as "increasing", "stable", "falling", that are easy to state in natural language but hard to specify using existing query languages. Moreover, pattern matching tasks often go beyond finding a sequence of patterns, requiring arbitrary combinations, e.g., disjunction, conjunction or quantification, with varying location or width constraints. Examples include finding stocks with at least 2 peaks within a span of 6 months, e.g., the so-called "double/triple top" patterns that indicate future downtrends [2], or finding cities where the temperature rises from November to January and falls during May to July such as Sydney.

Ad-hoc and Interactive Querying. Pattern-based queries are often defined *on-the-fly* during analysis, based on other patterns observed. For instance, biologists often search for a pattern in a group of genes similar to a pattern recently discovered in another group [11]. Similarly, astronomers monitor the shape of the luminosity trends of stars over time to search for and characterize new planetary objects (Figure 3c). For example, a dip in brightness often indicates a planetary object passing between the star and the telescope.

To address these issues, we developed ShapeSearch, described next.

3. OVERVIEW OF SHAPESEARCH

ShapeSearch provides powerful yet flexible mechanisms for users to search for trendline visualizations with a desired shape. In this section, we first present an overview of ShapeSearch along with user experience.

ShapeSearch supports an interactive interface for composing shape queries. Figure 4 depicts this interface, with an example query on genomics data discussed in the introduction. Here, the user is interested in searching for genes that get suppressed due to the influence of a drug, depicted by a specific shape in their gene expression—first rising, then going down, and finally rising again—with three patterns: up, down, and up, in a sequence. To search for this shape, the user first loads the dataset [6]) via form-based options on the left (Figure 4 Box 1), and then selects the space of visualizations to explore by setting the x axis as time, the y axis as expression values, and the category as gene. Each value of the category attribute results in a candidate visualization with the given x and y axis. Thus, the category attribute defines the space of visualizations over which we match the shape. ShapeSearch supports three mechanisms for shape specification—natural language, regular expressions (regex for short), and sketching on a canvas:

Sketching on Canvas. By drawing the desired shape as a sketch on the canvas (Figure 4 Box 2a), the user can search for visualizations that are *precisely* similar (using a distance measure such as Euclidean distance or Dynamic Time Warping [18]). As soon as the user finishes sketching, ShapeSearch outputs visualizations that are similar to the drawn sketch in the results panel (Figure 4 Box 4).

Natural Language (NL). For searching for visualizations that approximately match patterns, users can use natural language. For instance, as in Figure 4 Box 2b, the desired shape in the aforementioned genomics example can be expressed as *"show me genes that are rising, then going down, and then increasing"*. Similarly, scientists analyzing cosmological data can easily search for supernovae (bright stellar explosions) using *"find objects with a sharp peak in luminosity"*. We describe in Siddiqui et al. [26] how ShapeSearch translates natural language queries to a structured internal representation.

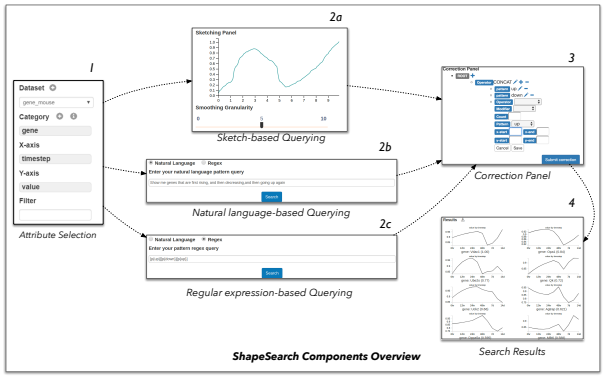


Figure 4: ShapeSearch Interface, consisting of six components. 1) Data upload, attribute selection, and applying filter constraints 2) Query specification: 2a) Sketching canvas 2b) Natural language query interface, and 2c) Regular expression interface, 3) Correction panel, and 4) Results panel

Regular Expression (regex). For queries that involve complex combinations of patterns that are difficult to express using natural language or sketch, the user can issue a regular expression-like query that directly maps to the structured internal representation, consisting of ShapeSearch primitives and operations, described in detail in Section 4.

During exploration, users can choose specification mechanisms interchangeably based on the complexity of the query. For both NL as well as regex, ShapeSearch additionally supports an autocomplete functionality to guide users towards their target query. We use the term *user query* to refer to the submitted query using any of the specification mechanisms.

The ShapeSearch back-end parses and translates the user query into a ShapeQuery, a structured internal representation of the query consisting of operators and primitives supported in our algebra (Section 4). The back-end supports an ambiguity resolver that uses a set of rules for automatically resolving syntactic and semantic ambiguities, as well as forwards the parsed query to the user for further corrections and validation (Figure 4 Box 3). The validated query is finally optimized and executed by the execution engine (Section 4.3), and the top visualizations that best match the ShapeQuery are presented to the user in the results panel (Figure 4 Box 4). Next, we discuss a ShapeQuery algebra that makes the core of ShapeSearch.

4. SHAPE ALGEBRA

ShapeQueries help express a large variety of patterns over trendlines with a minimal set of primitives and operators. A ShapeQuery represents a *shape* as a combination of multiple *simple patterns*. A simple pattern can either be precise with specific location constraints, e.g., matching $y = x$ between $x = 2$ to $x = 6$, or fuzzy, e.g., roughly increasing, where the notion of the pattern is approximate and its location unspecified. Each simple pattern along with its precise or imprecise constraints is called a ShapeSegment. Complex shapes, e.g., rising and then falling, are formed by combining multiple ShapeSegments using one or more *operators*. One can search for multiple patterns in a sequence (concat, $\otimes$) or matching the same sub-region of the trendline (and, $\odot$), or one of many patterns matching a sub-region (or, $\oplus$), described later.

As an example,"rising from x=2 to x=5 and then falling" can be translated into a ShapeQuery [x.s=2,x.e=5, p=up] $\otimes$[p=down] consisting of two ShapeSegments separated by a $\otimes$ operator. The first ShapeSegment captures "rising from $x = 2$ to $x = 5$"; the second expresses a "falling" pattern. Since the second must "follow"

Table 3: Primitives and Operators in ShapeQuery

Symbol	Name	Type
x.s	START X VALUE	Location Sub-Primitive
y.s	START Y VALUE	Location Sub-Primitive
x.e	END X VALUE	Location Sub-Primitive
y.e	END Y VALUE	Location Sub-Primitive
v	SKETCH	Location Sub-Primitive
.	ITERATOR	Location Sub-Primitive
p	PATTERN	Primitive
$	POSITION	Pattern Sub-Primitive
m	MODIFIER	Primitive
>	MORE	Modifier value
>2	ATLEAST 2X	Modifier value
=	SIMILAR	Modifier value
⊗	CONCAT	Operator
⊙	AND	Operator
⊕	OR	Operator
!	OPPOSITE	Operator

the first, the two ShapeSegments are combined using the CONCAT operator, denoted by ⊗. We now describe the shape primitives and operators that constitute the ShapeQuery algebra. Table 3 lists these primitives and operators.

4.1 Shape Primitives and Operators

A ShapeSegment is described using two high level primitives: LOCATION and PATTERN. The LOCATION values can be skipped in order to match the PATTERN anywhere in the trendline. Similarly, users can input the exact trendline to match, or the endpoints of the ShapeSegments to match without specifying the PATTERN.

Specifying LOCATION. LOCATION defines the endpoints of the sub-region of the trendline between which a pattern is matched: starting X/Y coordinate (x.s/y.s), ending X/Y coordinate (x.e/y.e). For example, [x.s=2,x.e=10, y.s=10, y.e=100] is a simple ShapeQuery to find trendlines whose trend between x=2 to x=10 is similar to the line segment from $(2,10)$ to $(10,100)$. Users can also draw a sketch to find trendlines similar to the sketch, a functionality supported in other tools alluded to in the introduction [7,14,23]. ShapeSearch translates the pixel values of the user-drawn sketch to the domain values of the X and Y attributes, and adds the transformed vector of (x,y) values as a vector v in the ShapeQuery. As an example, the ShapeQuery [v=(2:10,3:14,...,10:100)] finds trendlines that have precisely similar values to v using some distance measure, e.g., Euclidean distance, or dynamic time warping [18].

Specifying PATTERN. PATTERN defines a trend or a semantic feature in a sub-region of the trendline. A number of basic semantic patterns, commonly used for characterizing trendlines, are supported, such as *up*, *down*, *flat*, or the slope (θ) in degrees. For example [p=up] finds trendlines that are increasing, [p=45] finds trendlines that are increasing with a slope of about 45°, and [x.s=2,x.e=10, p=up] finds trendlines that are increasing from $x = 2$ to 10. Finally, one can use p=* to match any pattern and p=empty to ensure that there are no points over the sub-region.

Combining PATTERNs. ShapeQuery supports three operators to combine ShapeSegments:

- CONCAT (⊗) specifies a sequence of two or more ShapeSegments. For example, using [p=up]⊗[p=down] one can search for genes that are first rising, and then falling. Note that ⊗ is one of the most frequently used operations, and we sometimes omit ⊗ between ShapeSegments, e.g, [p=up][p =down], to make it succinct to describe.
- AND (⊙) simultaneously matches multiple patterns in the same sub-region of the trendline. Unlike CONCAT, all of the patterns must be present in the same sub-region. For example, one can look for genes whose expression values rise twice but do not fall more than once within the same sub-region.
- OR (⊕) searches for one among many patterns in the same sub-region of the trendline, picking the one that matches the sub-region best. For example, one can search for genes whose expressions are either *up*- or *down*-regulated.

Table 4: Pattern Scores

P	Score
up	$\frac{2 \cdot tan^{-1}(slope)}{\pi}$
down	$-\frac{2 \cdot tan^{-1}(slope)}{\pi}$
flat	$(1.0 - \|\frac{4 \cdot tan^{-1}(slope)}{\pi}\|)$
$\theta = x$	$(1.0 - \|\frac{2 \cdot tan^{-1}(slope-x))}{(\pi - \|tan^{-1}(x)\|)}\|)$
*	1
empty	-1
v	L_2 norm (configurable)

Table 5: Operator Scores

O	Score
⊗	$\sum_{i=1}^{k} score_i/k$
⊙	$min(score_1, ..., score_k)$
⊕	$max(score_1, ..., score_k)$

Comparing Patterns. In some cases, one may want to compare the pattern in a ShapeSegment with the preceding or succeeding ShapeSegments. To support such use cases, ShapeSearch (i) allows a ShapeSegment to refer to the previous or the next ShapeSegment using $+ or $− respectively, and (ii) compare patterns between the current and referred ShapeSegment using operations >, <, or =. For example, astronomers can issue a ShapeQuery [p=up]⊗[p < $−.p] with x=time and y=luminosity (brightness) to search for celestial objects that were initially moving rapidly towards earth, but after some point either slowed down or started moving away. [p < $−.p] ensures that the slope of brightness over time is less than that in the previous sub-region [p=up].

Expressing Complex Patterns. The aforementioned basic primitives and operators are powerful enough to express more complex ShapeSearch use-cases. We discuss three such complex patterns below, along with shortcuts for their easy specification.

1. Searching shapes of specific width. In some cases, users want to find specific shapes irrespective of their start location, e.g., searching for cities with the steepest rise in temperature over a width of 3 months. To express such queries, ShapeSearch supports the ITERATOR (.), e.g., [x.s=.,x.e=x.s+3, p=up] that iterates over all points in the trendline, setting each point as the start x position, with the x end position set to 3 units ahead. Internally, for a trendline of length n, this query can be rewritten as an OR operation over $(n-3+1)$ ShapeSegments, where, for the ith ShapeSegment, x.s=i and x.e=i+3.

2. Quantifiers. One can search for trendlines where a pattern occurs a specific number of times using quantifiers, denoted by q. For example, [p=up,q={1,2}] can be used to search for trendlines where there is an increasing pattern at least once and at most twice. Quantifiers can be internally rewritten using an OR of one or more CONCAT operations. For example, the above query is rewritten as ([p=*] ⊗[p=up] ⊗[p=*])⊕([p=*] ⊗[p=up]⊗[p=*] ⊗[p=up] ⊗[p=*]).

3. Nesting. A combination of patterns can be constrained to be within a specific sub-region by specifying them as a value of the PATTERN primitive. For example, to search for stocks that increased anytime between February to October, we can use nesting as follows: [x.s=2,x.e=10, p=([p=*] [p=up][p=*])]. This can be rewritten using CONCAT operations as follows: [x.s=2,p=*] ⊗[p=up]⊗[p=*]⊗[x.s=10,p=*].

4.2 Scoring

A ShapeQuery Q operates on one trendline, V_i, at a time, and returns a real number, called *score*, between −1 to +1. The ShapeQuery Q operates on V_i with the help of ShapeSegments $(S_1, S_2, \ldots, S_n)$ and operators $(O_1, O_2, \ldots, O_m)$. Each ShapeSegment S_i operates on $V_i^{p,q}$, a sub-region of V_i starting at $p = x.s$ and ending at $q = x.e$ and returns a $score_i \in [-1,1]$ using scoring functions we describe subsequently. One or more ShapeSegments are combined using operators such as ⊗, ⊙, ⊕. Formally, an operator O_i takes as input the scores $score_1, score_2, \ldots, score_n$ from its n input ShapeSegments and outputs another $score_i$ using scoring functions that capture the behavior of the operators.

For both efficiency and effectiveness, ShapeSearch approximates each sub-region with a line, using the slope to quantify how closely

it captures any given ShapeSegment. As depicted in Table 4, ShapeSearch uses different scoring functions for each pattern primitive that transforms the slope to a value in $[-1,1]$ using a tan^{-1} function. For example, for an *up* pattern, the function returns a score between $[0,1]$ for a trendline with a slope from $0°$ to $90°$, a score of $[-1,0]$ for a slope of less than $0°$ (opposite of *up*).

For execution, ShapeSearch takes the entire trendline, the Abstract Tree Representation (AST) of ShapeQuery, and the list of scoring functions *ScrFunc* as in Tables 4 and 5 as inputs. If the root node of the ShapeQuery tree is a ShapeSegment, ShapeSearch directly computes the score of ShapeSegment on the specified part of the trendline. If the root node is $\odot$ or $\oplus$, ShapeSearch invokes each of the operands (i.e., child sub-trees) to compute their scores on the sub-region independently, combining the scores as per operator's functions. However, if the root node is a CONCAT with k operands, i.e., child sub-trees, ShapeSearch segments L into all possible k sub-regions: $L_1, L_2, ..., L_k$ and then, for each segmentation, invokes the ith operand on ith segment. Finally, the maximum score across all segmentations is output.

4.3 Executing Fuzzy ShapeQueries

A common subclass of ShapeQueries are *fuzzy* ShapeQueries, consisting of at least one ShapeSegment with missing or multiple possible values for $x.s$ or $x.e$. Thus, for fuzzy ShapeQueries, we try all possible values of p and q, selecting the sub-region that leads to the best score. This becomes prohibitively expensive as the number of points in the trendline increases. For a CONCAT with k operands, the exhaustive approach creates $n^{(k-1)}$ segmentations, where n is the number of points in the trendline.

The Dynamic Programming Algorithm. We can show [26] that for the CONCAT operation, the scoring of the *jth* operand on *jth* sub-region does not depend on the scoring of the first $j-1$ operands on the first $j-1$ sub-regions. We use this idea to develop a faster dynamic programming algorithm (DP) for scoring CONCAT operations over ShapeSegments. Formally, let $OPT(1,t,(1:j-1))$ be the best score corresponding to the optimal segmentation over the sub-region between $x=1$ to $x=t$ for the first $j-1$ operands, and $SC(t+1,i,j)$ be the score of the *jth* operand over the sub-region between $x=t+1$ and $x=i$. Then, the optimal segmentation $OPT(1,i,(1:j))$ for the first j operands over $x=1$ and $x=i$ can be computed using the following recursion:

$$OPT(1,i,(1,j)) = \underset{t}{MAX}\{\frac{(j-1)\times OPT(1,t,(1:j-1))+SC(t+1,i,j))}{j}\}$$

Unfortunately, even though the DP algorithm is orders of magnitude faster than the exhaustive approach, for trendlines with large number of points, even a ShapeQuery with a single CONCAT operation can be slow, because of its quadratic runtime. We, next, discuss optimizations to further decrease the runtime of CONCAT operation on ShapeSegments.

4.3.1 A Pattern-Aware Bottom-up Approach

While the DP-based optimal approach scores all possible sub-regions for each operand in the CONCAT operation, a more efficient approach could be to select end points to be those where the slope (or pattern) changes drastically. We first illustrate our intuition, and then describe an algorithm that performs segmentation in a pattern-aware manner.

Intuition. As depicted in Figure 5, consider two sub-regions A on the left and B on the right for the trendline L. Say the trendline in sub-region A is inverted V-shaped, i.e., increasing until a point P and then decreasing. Now, for all possible segmentations where [p=up]'s sub-region lies completely in A, there are the following possibilities for x.e of [p=up]: 1) [p=up]'s x.e point is before P. 2) [p=up]'s x.e point is after P. 3) [p=up]'s x.e point is at P.

Since [p=down] follows [p=up], we can see that option 1 that sets [p=up]'s x.e $< P$ is less likely to be optimal as that will lead to scoring of a part of [p=down] on an increasing trend. Similarly,

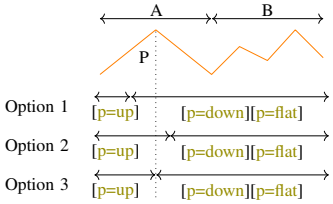


Figure 5: Pattern-aware selection of LOPs

x.e $> P$ is less optimal as that will lead to scoring of a part of [p=up] on a decreasing trend. Thus, if we have to (greedily) select one point in sub-region A for [p=up]'s x.e, P is likely a better choice. We call such a point as *locally optimal point* (LOP).

A Bottom-up Algorithm. Based on the above intuition, we develop a much faster algorithm that uses the following assumption to reduce the number of segmentations.

Assumption 4.1 (Closure)**.** *If a point is not locally optimal for any of the sub-expressions in the* CONCAT *operation (i.e., a* CONCAT *on a sub-sequence of the operands), it cannot be* x.s *or* x.e *of a* ShapeSegment *in the optimal segmentation.*

That is, local optimality leads to global optimality. Because of this assumption, our proposed algorithm is approximate. However, our empirical results show that despite this assumption, the accuracy of the algorithm is very close to that of DP, while taking orders of magnitude less time.

At a high level, the algorithm starts by dividing the trendline into smaller contiguous sub-regions. Next, it selects locally optimal points (LOPs) over small sub-regions, followed by a bottom-up merging step that uses LOPs over small sub-regions to find LOPs over larger sub-regions.

Selection of LOPs. We define a point P to be a LOP in a sub-region A for the sub-expression S_i if it is either the x.e of the first ShapeSegment *or* x.s of the last ShapeSegment of S_i. For instance, in the above example, it is easy to see that a LOP P in sub-region A is the x.e value of [p=up] in the optimal segmentation of [p=up]⊗[p=down] in A. Since a CONCAT operation with k operands can have (k^2) sub-sequences, there can be a maximum of $2.k^2$ LOPs in A.

Merging. Next, we incrementally merge nodes in a bottom-up fashion to select LOPs over larger sub-regions. For example, in Figure 6, node 4 depicts the sub-sequences formed by combining sub-sequences from nodes 1 and 2, and node 5 depicts the sub-sequences formed by combining sub-sequences from nodes 3 and 4. When multiple sub-sequences in the children nodes generate the same sub-sequence in the parent node, we select the one with maximum score after concatenation (i.e, the one with the most optimal segmentation), thereby pruning out LOPs corresponding to non-selected sub-sequences. For example, at node 5, a⊗b can be computed from 1) a from node 3 and b from node 4, 2) a⊗b from node 3 and b from node 4, and 3) a from node 3 and a⊗b from node 4. Among these 3 concatenations, we pick the one that gives the maximum score. This merging process is repeated at each intermediate node. Finally, at the root node, we select the points that result in the maximum score for the entire sequence of operands. More details along with the pseudo-code can be found in [4].

Given the closure assumption, we prove in [4] that the merging process leads to optimal segmentation and that the bottom-up algorithm with k CONCAT operands is optimal with a time complexity of $O(nk^4)$, i.e., linear in the number of points in the trendlines.

5. NATURAL LANGUAGE TRANSLATION

So far, we haven't described how natural language queries are parsed into ShapeQueries. We provide a brief overview of the three key steps involved in parsing, and refer readers to our extended report [4] for additional details. We use the following natu-

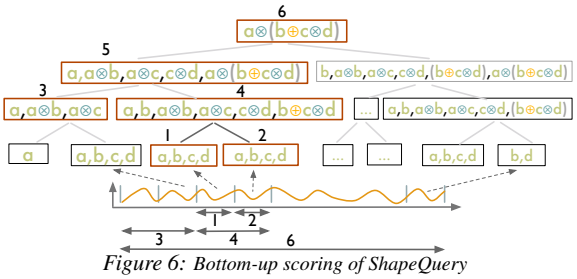


Figure 6: Bottom-up scoring of ShapeQuery

ral language query collected from MTurk for illustration: "show me the trendlines that are increasing from 2 to 5 and then decreasing".

Step 1. Primitives and Operators Recognition. Given a natural language query, the first step is to map words to their corresponding shape primitives and operators. For example, the above query is tagged as "show (noise) me (noise) the (noise) trendlines (noise) that (noise) are (noise) increasing (p) from (noise) 2 (x.s) to (noise) 5 (x.e) and then (⊗) decreasing (p)". In order to do so, we learn a linear-chain conditional-random field model (CRF) [10] and train it on the same 250 natural language queries we collected via Mechanical Turk (described in [4]) for understanding query characteristics. For each word, we use its part-of-speech (POS) tags along with word-level context as features.

Step 2. Identifying Pattern Value. For each of the words predicted of type p, e.g., increasing and decreasing in the above query, we additionally map them to the corresponding semantic pattern supported in ShapeSearch, e.g., "increasing" is mapped to p=up. For this mapping, ShapeSearch computes the similarity between the specified word and synonyms of the supported patterns, first using edit distance and then using wordnet [19]. The semantic pattern with the highest similarity between any of its synonyms and the specified word is selected.

Step 3. ShapeQuery Generation and Ambiguity Resolution. Next, we group primitives and operators into a ShapeQuery, first grouping all primitives between two operators into a single ShapeSegment. For the example query, the primitives are grouped as follows: [increasing (p=up), 2 (x.s), 5 (x.e)] and then (⊗) [decreasing (p=down)]. In some cases, this leads to incorrect grouping of primitives, e.g., two patterns in the same ShapeSegment. There could further be semantic ambiguity because of wrong entity tagging, e.g., decreasing (p=up) from 5 (y.s) to 10 (y.e) where x.s and x.e values are wrongly tagged as y.s and y.e respectively. ShapeSearch uses rule-based transformations that try to reorder and change the types of entities to get a correct and meaningful ShapeQuery [26].

The parsed ShapeQuery is sent to the front-end (Box 4 in Figure 4) for users to edit or further refine it if needed. The validated query is then executed to generate the matching trendlines.

6. FUTURE DIRECTIONS

We now discuss open research directions for improving the usability and performance of both Zenvisage and ShapeSearch.

6.1 Search Enumeration + Shape Matching

In ShapeSearch, users currently need to specify the X and Y attribute before issuing ShapeQueries. However, in certain scenarios, users may not know the X and Y attributes in advance or may want to search for the same shape over different combinations of attributes. Additionally, users may want to issue a multi-step query involving multiple shapes at the same time, finding states with decreasing listing prices trends but increasing soldprice trends of houses. To support such complex data exploration needs, we envision integrating ZQL with ShapeQuery. One simple option is support ShapeQuery as a functional primitive as part of the Process column in ZQL. For instance, Table 6 depicts an integrated query for the above example for finding states with decreasing listing prices trends but increasing soldprice trends. Combining ZQL and ShapeQuery also adds to expressiveness and efficiency of ZQL—functional primitives are currently are treated as black boxes and thus not optimized in Zenvisage. By adding support for ShapeQuery, Zenvisage can leverage the shape matching algorithms discussed earlier for efficient processing of visualizations.

6.2 In-Database Support for Fuzzy Matching

ShapeSearch performs shape matching outside relational databases; consequently as the size of the dataset increases, the data transfer and serialization/deserialization overheads tend to dominate, resulting in an increase in latency. On the other hand, recognizing patterns in a sequence of rows in relational databases has been widely desired but only supported by a few vendors. For instance, Oracle Database 12c supports a MATCH RECOGNIZE [3] clause for pattern matching in native SQL. SQL-TS (Simple Query Language for Time Series) [21] is another proposal on SQL extensions for pattern queries. Nevertheless, none of these extensions support fuzzy matching capabilities, instead they require users to define the patterns (e.g, up, down) using values of matching columns —making the specification quite tedious and verbose.

In order to support fuzzy shape queries, we envision developing new database extensions that take as input a ShapeQuery as part of the SQL query and leverage shape matching algorithms for efficiently executing the ShapeQuery within the database kernel. For instance, the following query depicts potential extensions for supporting ShapeQuery within the SQL syntax.

```
SELECT *
FROM Ticker T,
(MATCH BY symbol ON price
USING PATTERN [p=*]⊗p=down]⊗[p=up]⊗[p=*] AS score
ORDER BY score DESC
LIMIT 1) S
WHERE T.symbol = S.symbol
```

Given a table Ticker, the above query finds a stock symbol (specified via MATCH BY clause) with closest matching V-shaped trend on the values of column price (specified via ON clause) and outputs its corresponding tuples.

6.3 Supporting Context-Aware Search

During exploratory data analysis, users do not always have a precise pattern query to start with, instead they compose queries as the exploration evolves. Often, they break their pattern queries into a sequence of simpler queries that build upon prior ones. Thus, for a more fluid user-experience, there are interesting avenues for future work that capture context, suggest next steps, and support incremental composability. One option is to make meaningful query suggestions by mining query patterns from past search logs and match them with immediately preceding queries (i.e., the context) in the same session. Since there can be a large number of patterns, efficiently searching for prefixes while effectively capturing user's intent is an interesting challenge.

For incremental composability, the context of user exploration can be represented using a state machine consisting of partial queries as different possible states. The state can then be incrementally updated as new queries arrive. For novice analysts, interactive querying interfaces, similar to systems such as GestureDB [16] and DataPlay [5] can help make query specification even easier. In addition, like in Zenvisage, ShapeSearch can be extended to automatically find typical and outliers patterns to help the users get started quickly.

6.4 Mixing Code and Interaction

To accelerate the process of data exploration, another important next step is to integrate visualization search abstractions supported

Name	X	Y	Z	Process
f1	year	listprice	z1 <– 'state'.*	z2 <– $argany_{z1}$[p=down]($f1$)
f2	year	soldprice	z1	z3 <– $argany_{z1}$[p=up]($f2$)
*f3	year	{listprice, soldprice }	z2 && z3	

Table 6: Example of a ZQL *query using* ShapeQuery *as a functional primtive within Process column. The query finds states with decreasing listprice but increasing sold price over trends of houses.*

via ZQL and ShapeQuery with existing data science libraries such as Pandas. This will allow users to seamlessly transition between writing code (e.g., for data edits, cleaning, and transformation); getting recommendations via search specifications, and performing interactions on visualizations—all in one place. As a step in this direction, LUX [1], a recent Python library, combines partial user-specifications with best practices from visual data analysis to recommend interesting visualizations for guiding users towards next steps. It further displays visualizations as a widget in-situ within a Jupyter notebook to support easier transitions between code and interaction. While specification in LUX is inspired from ZQL, adding natural-language or regex-based pattern searching functionalities, as supported in ShapeQuery, can further enhance the power of such libraries.

7. RELATED WORK

Our work draws on prior work in visual querying, as well as symbolic pattern mining. Visual querying tools [14, 15, 20, 23, 27] help users search for visualizations with a desired shape by taking as input a sketch of that shape. Most of these tools perform precise point-wise matching using measures such as Euclidean distance or DTW. A few tools such as TimeSearcher [7] let users apply soft or hard constraints on the x and y range values via boxes or query envelopes, but do not support mechanisms for specifying shape primitives beyond location constraints. ShapeSearch introduces a novel algebra that improves extensibility by acting as a common "substrate" for various input mechanisms, along with an optimization engine that efficiently matches patterns against a large collection of trendlines.

Symbolic sequence matching papers approach the problem of pattern matching by employing offline computation to chunk trendlines into fixed length blocks, encoding each block with a symbol that describes the pattern in that block [8, 9, 12, 17, 22]. Since these work operates on pre-chunked-and-labeled trendlines, the problem is one of matching regular expressions against string sequences (one per pre-labeled trendline). Most of these papers only return a boolean score for whether the pattern matches the string sequence. Moreover, since the trendlines are pre-labeled and indexed, they do not support on-the-fly pattern matching where the same trendline can change shapes based on filters or aggregation constraints. ShapeSearch, on the other hand, adopts a more online query-aware ranking of trendlines without requiring precomputation, and is thus more suited for ad-hoc data exploration scenarios.

8. CONCLUSION

In this work, we described ShapeSearch, a pattern matching system that complements our prior system Zenvisage by providing expressive and flexible mechanisms for domain experts to effortlessly and efficiently search for trendline visualizations. We described the ShapeQuery algebra that forms the core of ShapeSearch, and helps express a large variety of patterns with a minimal set of primitives and operators. The algebra is backed by a shape matching engine that enables on-the-fly and scalable pattern matching. Overall, together with Zenvisage, ShapeSearch offers a promising first step towards substantially simplifying and improving the process of interactive data exploration for novice and expert analysts alike.

9. REFERENCES

[1] https://github.com/lux-org/lux. https://github.com/lux-org/lux.
[2] Investopedia. https://www.investopedia.com/terms/t/tripletop.asp.
[3] Match recognize. [https://bit.ly/3bWwUhs. Online; accessed 17-Aug-2015].
[4] Technical report. https://arxiv.org/abs/1811.07977.
[5] A. Abouzied, J. M. Hellerstein, and A. Silberschatz. Playful query specification with dataplay. *PVLDB*, 5(12):1938–1941, 2012.
[6] C. J. Bult, J. T. Eppig, J. A. Kadin, J. E. Richardson, J. A. Blake, and M. G. D. Group. The mouse genome database (mgd): mouse biology and model systems. *Nucleic acids research*, 36(suppl_1):D724–D728, 2008.
[7] P. Buono, A. Aris, C. Plaisant, A. Khella, and B. Shneiderman. Interactive pattern search in time series. In *Visualization and Data Analysis 2005*, volume 5669, pages 175–187. International Society for Optics and Photonics, 2005.
[8] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos. *Fast subsequence matching in time-series databases*, volume 23. ACM, 1994.
[9] M. N. Garofalakis, R. Rastogi, and K. Shim. Spirit: Sequential pattern mining with regular expression constraints. In *VLDB*, volume 99, pages 7–10, 1999.
[10] J. Lafferty, A. McCallum, and F. C. Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. 2001.
[11] D. J.-L. Lee, J. Lee, T. Siddiqui, J. Kim, K. Karahalios, and A. Parameswaran. You can't always sketch what you want: Understanding sensemaking in visual query systems. *IEEE transactions on visualization and computer graphics*, 2019.
[12] R. A. K.-l. Lin and H. S. S. K. Shim. Fast similarity search in the presence of noise, scaling, and translation in time-series databases. In *Proceeding of the 21th International Conference on Very Large Data Bases*, pages 490–501. Citeseer, 1995.
[13] M. Mannino and A. Abouzied. Expressive time series querying with hand-drawn scale-free sketches. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, page 388. ACM, 2018.
[14] M. Mohebbi, D. Vanderkam, J. Kodysh, R. Schonberger, H. Choi, and S. Kumar. Google correlate whitepaper. 2011.
[15] P. K. e. a. Muthumanickam. Shape grammar extraction for efficient query-by-sketch pattern matching in long time series. In *Visual Analytics Science and Technology (VAST), 2016 IEEE Conference on*, pages 121–130. IEEE, 2016.
[16] A. Nandi, L. Jiang, and M. Mandel. Gestural query specification. *PVLDB*, 7(4):289–300, 2013.
[17] R. A. G. Psaila and E. L. Wimmers Mohamed &It. Querying shapes of histories. *Very Large Data Bases. Zurich, Switzerland: IEEE*, 1995.
[18] L. Rabiner, A. Rosenberg, and S. Levinson. Considerations in dynamic time warping algorithms for discrete word recognition. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 26(6):575–582, 1978.
[19] R. P. Roetter, C. T. Hoanh, A. G. Laborte, H. Van Keulen, M. K. Van Ittersum, C. Dreiser, C. A. Van Diepen, N. De Ridder, and H. Van Laar. Integration of systems network (sysnet) tools for regional land use scenario analysis in asia. *Environmental Modelling & Software*, 20(3):291–307, 2005.
[20] K. Ryall, N. Lesh, T. Lanning, D. Leigh, H. Miyashita, and S. Makino. Querylines: approximate query for visual browsing. In *CHI'05 Extended Abstracts*, pages 1765–1768, 2005.
[21] R. Sadri, C. Zaniolo, A. Zarkesh, and J. Adibi. Expressing and optimizing sequence queries in database systems. *ACM Transactions on Database Systems (TODS)*, 29(2):282–318, 2004.
[22] H. Shatkay and S. B. Zdonik. Approximate queries and representations for large data sequences. In *Data Engineering, 1996. Proceedings of the Twelfth International Conference on*, pages 536–545. IEEE, 1996.
[23] T. Siddiqui, A. Kim, J. Lee, K. Karahalios, and A. Parameswaran. Effortless data exploration with zenvisage: an expressive and interactive visual analytics system. *Proceedings of the VLDB Endowment*, 10(4):457–468, 2016.
[24] T. Siddiqui, J. Lee, A. Kim, E. Xue, X. Yu, S. Zou, L. Guo, C. Liu, C. Wang, K. Karahalios, et al. Fast-forwarding to desired visualizations with zenvisage. In *CIDR*, 2017.
[25] T. Siddiqui, P. Luh, Z. Wang, K. Karahalios, and A. Parameswaran. Shapesearch: flexible pattern-based querying of trend line visualizations. *Proceedings of the VLDB Endowment*, 11(12):1962–1965, 2018.
[26] T. Siddiqui, P. Luh, Z. Wang, K. Karahalios, and A. Parameswaran. Shapesearch: A flexible and efficient system for shape-based exploration of trendlines. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 51–65, 2020.
[27] M. Wattenberg. Sketching a graph to query a time-series database. In *CHI'01 Extended Abstracts on Human factors in Computing Systems*, pages 381–382. ACM, 2001.

Technical Perspective: Optimistically Compressed Hash Tables & Strings in the USSR

Marcin Zukowski
Snowflake Inc.
marcin.zukowski@snowflake.com

Hash tables are possibly the single most researched element of the database query processing layers. There are many good reasons for that. They are critical for some key operations like joins and aggregation, and as such are one of the largest contributors to the overall query performance. Their efficiency is heavily impacted by variations of workloads, hardware and implementation, leading to many research opportunities. At the same time, they are sufficiently small and local in scope, allowing a starting researcher, or even a student, to understand them and contribute novel ideas. And benchmark them... Oh, the benchmarks... :)

This paper by Tim Gubner, Viktor Leis and Peter Boncz addresses less frequently researched aspects of hash tables, in particular string processing, and presents some useful real-system implementation ideas. It improves the main aspects of the (in-memory) hash table performance, CPU operations and memory accesses, with 3 techniques.

Domain-Guided Prefix Compression (DGPC), a combination of FOR coding and bit packing, is used to reduce the memory footprint. While similar methods have been proposed at the scan level, authors discuss how the range metadata can be propagated up and used in the higher layers of the query. This technique, while not discussed much in literature, is used by some systems and allows various optimizations. For example, the discussed idea of using this info to avoid the overflow checks is beneficial for both interpreted and compiled systems. In the case of this paper, DGPC significantly improves the memory usage and hence cache efficiency, with negligible processing overhead. Unfortunately, for optimal performance, the implementation complexity is far from trivial, esp. in a not-compiled engine.

Optimistic Splitting proposes dividing the per-row data stored in the hash table into a frequently accessed hot-set, and rarely accessed cold-set. The authors present a few concrete ideas for such a split, for example overflows in aggregations. This idea can be generalized to other techniques that pull information from later stages to earlier-accessed data structures. I am sure we will see additional ideas in this space in the future. Importantly, techniques like this can have a *negative* effect on performance, so triggering them in a safe and robust manner is very important.

The final technique, **Unique Strings Self-aligned Region (USSR)**, besides having a cool name, provides a simple but powerful mechanism that can improve string processing performance without the complexity of maintaining a full, global string dictionary. USSR does all that **without changing the data format** (assuming strings are kept as pointers). The USSR-aware operations can easily and quickly determine which strings belong to the USSR Data Region and optimize for them. This can dramatically simplify the complexity of implementing this in an existing system. At the same time, the price paid here is that USSR is **partial** and, while beneficial in many cases, in some situations it will not work.

If we treat the self-aligned block as the main building idea behind USSR, the implementation from this paper can be seen as one of the many choices from a broad design space:

- All strings stored in USSR are unique. An implementation without this can still provide some benefits
- A single USSR structure is used for all columns. With, e.g., per-attribute structures, some additional optimizations are possible (e.g., narrower codes for low cardinality key columns, useful for DGPC).
- The paper tightly couples the Data Region and the supporting Linear Hash Table structure, but other options could be used for the latter structure.
- Data Region size is fixed at 512KB. It is possible to envision different sizes, or even dynamic sizing.
- USSR stores strings 8-byte aligned and preceded by hash values. Both of these reduce space utilization and are not strictly needed.

USSR presents some additional opportunities, common to dictionary-compressed strings, that are not discussed in the paper. For example, USSR-strings can use faster serialization methods than other, non-persistent, strings. Another opportunity is fast memoization of various operations for USSR-strings during processing.

While simple in many aspects, USSR also introduces some implementation challenges. For example, some database operations (e.g., serializing data to disk or network) might not produce original string pointers, removing the ability to benefit from USSR-strings further in the query plan. Additionally, maintaining the USSR encoding can be tricky in parallel and, especially, distributed systems. While all these challenges can be addressed, they can significantly add to the implementation complexity.

In summary, if you are a database researcher or, especially, a database system engineer, *do* study this paper. Problems discussed here are real, the improvement ideas are valuable not only for hash table processing, and some presented techniques are (relatively) easy to add to an existing system.

Optimistically Compressed Hash Tables & Strings in the USSR

Tim Gubner
CWI
tim.gubner@cwi.nl

Viktor Leis
FSU Jena
viktor.leis@uni-jena.de

Peter Boncz
CWI
boncz@cwi.nl

ABSTRACT

Modern query engines rely heavily on hash tables for query processing. Overall query performance and memory footprint is often determined by how hash tables and the tuples within them are represented. In this work, we propose three complementary techniques to improve this representation: *Domain-Guided Prefix Suppression* bit-packs keys and values tightly to reduce hash table record width. *Optimistic Splitting* decomposes values (and operations on them) into (operations on) frequently- and infrequently-accessed value slices. By removing the infrequently-accessed value slices from the hash table record, it improves cache locality. The *Unique Strings Self-aligned Region* (USSR) accelerates handling frequently occurring strings, which are widespread in real-world data sets, by creating an on-the-fly dictionary of the most frequent strings. This allows executing many string operations with integer logic and reduces memory pressure.

We integrated these techniques into Vectorwise. On the TPC-H benchmark, our approach reduces peak memory consumption by 2–4× and improves performance by up to 1.5×. On a real-world BI workload, we measured a 2× improvement in performance and in micro-benchmarks we observed speedups of up to 25×.

1. INTRODUCTION

In modern query engines, many important operators like join and group-by are based on in-memory hash tables. Hash joins, for example, are usually implemented by materializing the whole inner (build) relation into a hash table. Hash tables are therefore often large and determine the peak memory consumption of a query. Since hash table sizes often exceed the capacity of the CPU cache, memory latency or bandwidth become the performance bottleneck in query processing. Due to the complex cache-hierarchy of modern CPUs, the access time to a random tuple varies by orders of magnitude depending on the size of the working set. This means that shrinking hash tables does not only reduce memory consumption but also has the potential of improving query performance through better cache utilization [4,5,21].

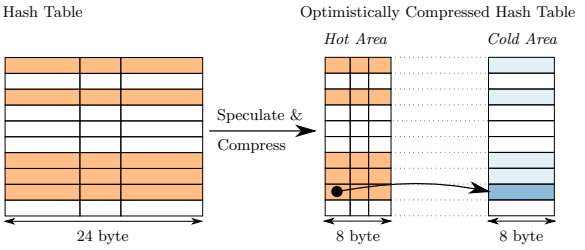


Figure 1: Optimistically Compressed Hash Table, which is split into a thin hot area and a cold area for exceptions

To decrease a hash table's hunger for memory and, consequently, increase cache efficiency, one can combine two orthogonal approaches: to increase the fill factor and to reduce the bucket/row size. Several hash table designs like Robin Hood Hashing [9], Cuckoo Hashing [19], and the Concise Hash Table [5] have been proposed for achieving high fill factors, while still providing good lookup performance. Here, we investigate how the size of each row can be reduced—a topic that, despite its obvious importance for query processing, has not received as much attention.

While heavyweight compression schemes tend to result in larger space savings, they also have a high CPU overhead, which often cannot be amortized by improved cache locality. Therefore, we propose a lightweight compression technique called *Domain-Guided Prefix Suppression*. It saves space by using domain information to re-pack multiple columns in-flight in the query pipeline into much fewer machine words. For each attribute, this requires only a handful of simple bit-wise operations, which are easily expressible in SIMD instructions, resulting in an extremely low per-tuple cost (sub-cycle) for packing and subsequent unpacking operations.

Rather than saving space, the second technique, called *Optimistic Splitting*, aims at improving cache locality. As Figure 1 illustrates, it splits the hash table into a hot (frequently accessed) and a cold (infrequently accessed) area containing exceptions. These exceptions may, for example, be overflow bits in aggregations, or pointers to string data. By separating hot from cold information, Optimistic Splitting improves cache utilization even further.

An often ignored, yet costly part of query processing is string handling. String data is very common in many real-world applications [18, 24]. In comparison with integers, strings occupy more space, are *much slower* to process, and are less amenable to acceleration using SIMD. To speed up

©IEEE 2020. This is a minor revision of the paper entitled "Efficient Query Processing with Optimistically Compressed Hash Tables & Strings in the USSR" published in the Proceedings of the 2020 ICDE Conference, 2375-026X/20.
DOI: 10.1109/ICDE48307.2020.00033

string processing, we therefore propose the *Unique Strings Self-aligned Region (USSR)*, an efficient dynamic string dictionary for frequently occurring strings. In contrast to conventional per-column dictionaries used in storage, the USSR is created anew for each query by inserting frequently occurring strings at query runtime. The USSR, which has a fixed size of 768 kB (ensuring its cache residency), speeds up common string operations like hashing and equality checks.

While each of the proposed techniques may appear simple in isolation, they are highly complementary. For example, using all three techniques, strings from a low-cardinality domain (few distinct strings) can be represented using only a small number of bits. Furthermore, our techniques remap operations on wide columns into operations on multiple thin columns using a few extra primitive functions (such as pack and unpack). As such, they can easily be integrated into existing systems and do not require extensive modifications of query processing algorithms or hash table implementations.

Rather than merely implementing our approach as a prototype, we fully integrated it into Vectorwise which originated from the MonetDB/X100 project [8]. Describing how to integrate the three techniques into industrial-strength database systems is a key contribution of the paper.

Based on Vectorwise, we performed an extensive experimental evaluation using TPC-H, micro-benchmarks, and real-world workloads. On TPC-H, we reduce peak memory consumption by 2–4× and improve performance by up to 1.5×. On a string-heavy real-world BI workload, we measured a 2.2× improvement in performance, and in micro-benchmarks we even observed speedups of up to 25×.

2. DOMAIN-GUIDED PREFIX SUPPRESSION

Domain-Guided Prefix Suppression reduces memory consumption by eliminating the unnecessary prefix bits of each attribute. This enables us to cheaply compress rows without affecting the implementation of the hash table itself, which makes it easy to integrate our technique into existing database systems. In particular, while our system (Vectorwise) uses a single-table hash join, Domain-Guided Prefix Suppression would also be applicable (and highly beneficial) for systems that use partitioning joins [4, 22]. Domain-Guided Prefix Suppression also allows comparisons of compressed values without requiring decompression.

2.1 Domain Derivation

A column in-flight in a query plan can originate directly from a table scan or from a computation. If a value originates from a table scan, we determine its domain based on the scanned blocks. For each block we utilize per-column minimum and maximum information (called ZoneMaps or Min/Max indices). This information is typically *not* stored inside the block itself as this would require scanning the block (potentially fetching it from disk) before this information can be extracted. Instead, the meta-data is stored "out-of-band" (e.g. in a row-group header, file footer or inside the catalog). By knowing the range of blocks that will be scanned, the domain can be calculated by computing the total minimum/maximum over the range.

On the other hand, if a value stems from a computation, the domain minimum and maximum can be derived bottom up according to the functions used, based on the minimum/maximum bounds on its inputs under the assumption of the worst case. Consider, for example, the addition of two integers $a \in [a_{\min}, a_{\max}]$ and $b \in [b_{\min}, b_{\max}]$ resulting in $r \in [r_{\min}, r_{\max}]$. To calculate $r_{\min}$ and $r_{\max}$ we have to assume the worst-case that means the smallest (r_{min}), respective highest ($r_{\max}$), the result of the addition. In case of an addition this boils down to $r_{\min} = a_{\min} + b_{\min}$ and $r_{\max} = a_{\max} + b_{\max}$.

Depending on these domain bounds, an addition of two 32-bit integer expressions could still fit in a 32-bit result, or less likely, would have to be extended to 64-bit. This analysis of minimum/maximum bounds often can allow implementations to ignore overflow handling, as the choice of data types *prevents* overflow, rather than having to *check* for it. For aggregation functions such as `SUM`, overflow avoidance is more challenging. In Section 3, we discuss Optimistic Splitting, which allows to do most calculations on small data types, also reducing the cache footprint of aggregates.

2.2 Prefix Suppression

Using the derived domain bounds, we can represent values compactly without losing any information by dropping the common prefix bits. To further reduce the number of bits and enable the compression of negative values, we first subtract the domain minimum from each value. Consequently, each bit-packed value is a positive offset to the domain minimum. We also pack multiple columns together such that the packed result fits a machine word. This is done by concatenating all compressed bit-strings and (if necessary) chunk the result into multiple machine words. Each chunk of the result constitutes a compressed column which can be stored just like a regular uncompressed column.

2.3 Compression and Decompression

Like many modern column-oriented systems, Vectorwise is based on vectorized *primitives* that process cache-resident vectors (=arrays of single column values). These primitives process items from multiple inputs in a data-parallel (SIMD-friendly) fashion in a tight loop. Consequently, modern compilers automatically translate such code into SIMD instructions for the specified target architecture (e.g. AVX-512). In our vectorized hash table implementation, `pack` primitives compress and "glue" multiple inputs together to produce one intermediate result. Later, this intermediate result is then stored inside the hash table. With all the inputs and the one output being cache-resident vectors, the compression itself happens in-cache. For bit-packing, our `pack` primitives look similar to the following pseudo-code:

```
void pack2_i32_i16_to_i32(i32* res, int n,
    i32* col1, i32 b1, int ishl1, int oshr1, i32 m1,
    i16* col2, i16 b2, int ishl2, int oshr2, i32 m2) {
  for (int i=0; i<n; i++) {
    // Select portion of input and cast to result's type
    i32 c1 = ((col1[i] - b1) >> ishl1) & m1;
    i32 c2 = ((col2[i] - b2) >> ishl2) & m2;
    // Move to output positions
    res[i] = (c1 << oshr1) | (c2 << oshr2);
} }
```

After bit-packing, we scatter the intermediate results into its final positions in the hash table. For improved cache locality, the hash table is stored in row-wise layout (NSM) [27].

When decompressing values, we fetch up to 4 columns from the hash table and directly decompress them. For decompressing a vector of `n` packed 16-bit integers from 32-bit

and 16-bit integers at positions `idx` in the hash table, this leads to the following pseudocode (2-column example):

```
void unpack2_i32_i16_to_i16(i16* res, int n, int* idx, i16 b,
    i32* col1, int ishr1, int oshl1, i16 m1, int s1,
    i16* col2, int ishr2, int oshl2, i16 m2, int s2) {
  for (int i=0; i<n; i++) {
    // DSM (columnar) position -> NSM (row) position
    int idx1 = idx[i] * s1;
    int idx2 = idx[i] * s2;
    // Extract relevant bits from NSM record
    i16 c1 = (col1[idx1] >> ishr1) & m1;
    i16 c2 = (col2[idx2] >> ishr2) & m2;
    // Stitch back together
    res[i] = (c1 << oshl1) | (c2 << oshl2) + b;
} }
```

Notably compression and decompression operate in a non-intuitive fashion: Both process m inputs and produce *one* output. This particular approach has two advantages: (a) In contrast to approaches with multiple outputs, it allows decompressing specific columns without enforcing decompression of neighboring cells. This allows an efficient mix of key checks on compressed data together with key checks on bit-packed non-integer data, most notably strings. (b) We concatenate bit-strings directly in registers, as opposed to approaches that partially compress/decompress which require multiple rounds of reading/writing from/to output vectors to concatenate partial output vectors into the final output.

2.4 Operating on Compressed Keys

Domain-Guided Prefix Suppression also allows comparing compressed values themselves (without having to decompress). Assume key value A is stored in the hash table and probe key B is compared to A. Normally one would just fetch the key A from the table and then compare it to B. In combination with compression, fetching A also requires decompressing A. We argue it is better to first bring B into the same representation as A, i.e., compressing B, and then directly compare the compressed values. This is especially true if keys A and B consist of multiple columns. For instance, a group-by on two columns can often be mapped into single-integer compressed key, reducing computational work (e.g. perform a single comparison, using fewer branches).

3. OPTIMISTIC SPLITTING

The goal of *Optimistic Splitting* is to exploit skewed access frequencies by separating the common case from exceptional situations. We physically split the hash table into two areas: The frequently-accessed *hot area* and the *cold area*, which is accessed rarely. This approach does not necessarily save space. However it shrinks the active working set, leading to lower memory access cost. Also, it converts operations on the final, widest, data type into operations on a potentially smaller data type. Specifically, if 128-bit operations become 64-bit or 32-bit; this can speed up computation noticeably. As we show in the following, Optimistic Splitting is especially important for data that is hard to compress such as aggregates and strings.

Aggregates are hard to compress with Domain-Guided Prefix Suppression as it is not possible to obtain tight bounds for aggregation results (for example `SUM`s). The reason is that one has to be pessimistic when deriving domain bounds to prevent integer overflows: Assuming a `SUM` of at most 2^{48} integers from, say, a 18-bit domain, would overflow 64-bit and thus need a 128-bit aggregate. If this type is used for the aggregate, on each addition in the sum this large 128-bit integer will be read, updated, and written back.

Table 1: Optimistic Aggregates

Aggregate	Common case	Exception
`SUM`	Small integer	Overflow counter
`MIN`	Small upper bound	Minimum
`MAX`	Small lower bound	Maximum
`COUNT`	Similar to `SUM`	
`AVG`	Rewritten into $\frac{\texttt{SUM}}{\texttt{COUNT}}$	

Using a 64-bit integer for the aggregate, on the other hand, would (a) reduce reads and writes by a factor 2 and (b) provide faster updates. Without sacrificing correctness, Optimistic Splitting allows one to do just that in the common case (i.e., when no overflow occurs): The 128-bit aggregate result is split into a frequently-accessed 64-bit sum and another, rarely-accessed 64-bit overflow/carry field, which is stored separately. In pseudocode, this looks as follows:

```
void opsum(u64* common, u64* except, int group, i32 value) {
  common[group] += value; // 64-bit unsigned addition
  // Overflow check
  bool overflow = common[group] < (u64)value;
  bool positive = value >= 0;
  if (!(overflow ^ positive)) { // Rare: handle overflow
    if (positive) except[group]++;
    else          except[group]--;
} }
```

Note that this is a generic implementation that handles positive as well as negative `values`. In combination with domain bounds (Min/Max information) it is possible to prove the absence of negative or positive values which leads to simplified logic and improved performance.

Similarly, it is possible to shrink the working set of other aggregates. Table 1 illustrates how to exploit Optimistic Splitting for these. We use the *associativity* of aggregates to provide a fast path for large aggregates and a smaller working set. `MIN` can be implemented using an upper bound (s) inside the hash table and storing the full minimum e as an exception ($s \geq e$). When calculating the aggregate, we first check against s and discard values that cannot become the new minimum. For the remaining values, we check against the full minimum and potentially update the full minimum e as well as the upper bound s. Similar is the implementation of `MAX` whereas the other aggregate functions, `COUNT` and `AVG`, can be implemented similar to `SUM`. However, in case of `COUNT` we can more aggressively reduce the common case to a 16-bit integer and after $2^{16} - 1$ iterations update both, the small optimistic counter as well as the exception.

Other Applications. Optimistic Splitting is a very general idea that we believe can be applied in many different use cases. It only requires that the entries of a hash table have different access patterns, and can be decomposed.

4. USSR: A DYNAMIC DICTIONARY

Strings are prevalent in many real-world data sets [13, 18, 24] and present additional challenges for query performance. In contrast to integers, any individual string generally does not fit into a single CPU register and requires multiple instructions for each primitive operation (e.g. comparison). Strings are also often larger than integers, which negatively affects memory footprint and cache locality. Furthermore, neither Domain-Guided Prefix Suppression nor Optimistic Splitting can directly be applied to strings. This section presents a dynamic data structure called *Unique Strings*

Self-aligned Region, which saves memory and enables processing strings at almost the same speed as integers.

4.1 The Problems with Global Dictionaries

To improve the performance of strings, some main-memory database systems—most notably SAP HANA [11]—represent strings using per-column dictionaries where codes respect the value order. Using these dictionaries, string comparisons and hashing operations can be directly performed on the dictionary keys, which are fixed-size integers, rather than variable-length strings. Unfortunately, global dictionaries have significant downsides, which have precluded their general adoption. First, because random access to the dictionaries is common, the dictionaries must fully reside in main memory. For systems that must manage data sets larger than main memory (e.g. analytical column stores), this is a major problem. Also, systems that support parallel and distributed execution, including those designed or optimized for the cloud, face the problem that bulk loading or updating tables in parallel would require continuous synchronization in order to maintain a consistent global dictionary. Another downside is that dictionaries incur significant overhead for inserts, updates, and deletes—in effect they are a mandatory secondary index on every string column. If, for instance, new values appear, extending the dictionary such that one additional bit is needed to represent a code, updates will no longer fit in previously encoded data. Deletes of no longer used strings leave holes in the code space that need to be garbage collected and inserts in sorted dictionaries often require re-coding (periodically rewriting all encoded columns).

Given these problems with global dictionaries, most database systems therefore limit themselves to per-block dictionaries (e.g. one dictionary for every 10,000 strings). With this approach, dictionaries are a local feature, mainly used for compression rather than a global data structure. Per-block dictionaries are often almost as space-effective as per-column dictionaries without sharing their in-memory limitations and update overheads. For query processing, however, the advantage of per-block dictionaries is limited. While some systems evaluate pushed-down selections directly on the dictionary [14], all other operations require decompression and therefore do not benefit from the dictionary. The reason is that the dictionary is only available to the table scan operators. Materializing operators like hash join and group-by, therefore, typically allocate heap memory on the heap for every string. Needless to say, this is very inefficient, yet dealing with strings is only a sparsely researched topic.

4.2 Unique Strings Self-aligned Region (USSR)

The USSR is a query-wide data structure that contains the common strings of a particular query. In contrast to the heap, all strings within the USSR are known to be unique, which enables fast operations on these strings. To make it cache resident and efficient, the USSR has a limited size. Once it is full, strings need to be allocated on the heap as usual. By removing duplicates in this opportunistic fashion, the USSR reduces the number of heap allocations and therefore minimizes peak memory consumption.

By default both, heap-backed and USSR-backed, strings are represented as normal pointers, which means that query engine operators can treat all strings uniformly without any code modifications. This allows to retro-fit this idea easily into already existing engines. However, by exploiting

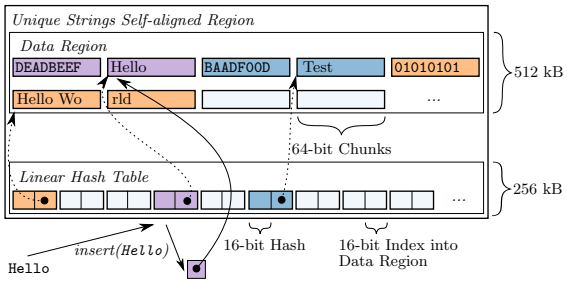


Figure 2: USSR data structure details

the dictionary-like nature and artful implementation of the USSR, the following additional optimizations become possible for USSR-based strings: (a) String comparisons are almost as fast as integer comparisons. (b) Hashes are precalculated and stored within the USSR, speeding up hash-based operators like join and group-by. (c) Since the size of the USSR is limited, frequent strings can also be represented using small integer offsets, which can be exploited e.g. in Optimistic Splitting.

To summarize, the USSR is a lightweight, dynamic, and opportunistic string dictionary. It does not require changes to the storage level, but is implemented in the query processor, and speeds up queries with low to medium string cardinalities, which is where global string dictionaries excel.

4.3 Data Structure Details

Our USSR implementation limits its capacity to 768 kB: it consists of a *hash table* (256 kB) and a *data region* (512 kB). Figure 2 serves as an illustration of the USSR.

The 512 kB data region starts at a *self-aligned* memory address (i.e., the pointer has 0s in its lowest 19 bits). If one allocates 1 MB of data, there is always a self-aligned address in its first half for the data region; and there is always either 256 kB space before or after the data region for the hash table. The self-aligned memory address guarantees that all pointers inside the data region start with the same 45-bit prefix. This allows to very efficiently test whether a string pointer points inside the USSR (by applying a mask).

The data region stores the string data and materializes the string's hash value just before it. These numbers are stored aligned, so the data region effectively consists of 64k slots of 8 bytes where a string can start. Given that each string takes at least two slots (one for the hash and one for the string) the USSR can contain maximally 32k strings.

When inserting a string, the USSR needs to check whether that string is already stored, and if so, return its address rather than insert a new string. To do this in low $O(1)$, there is a fast linear probing hash table, consisting of 64k 4-byte buckets. Each bucket consists of a 16-bit hash extract and a 16-bit *slot number* that points into the data region to the start of the string. The lowest 16-bits of the string hash are used for locating the bucket, and the next 16-bits are the extract used to quickly identify collisions. The load factor is always below 50% (64k buckets for at most 32k strings).

4.4 Insertion

The purpose of the USSR is to accelerate operations on frequent strings. In the extreme, all strings could be part of the USSR. However, due to its limited size, the USSR can only fit a sample. The sampling happens during insertion into the data structure. Failure during insertion might hap-

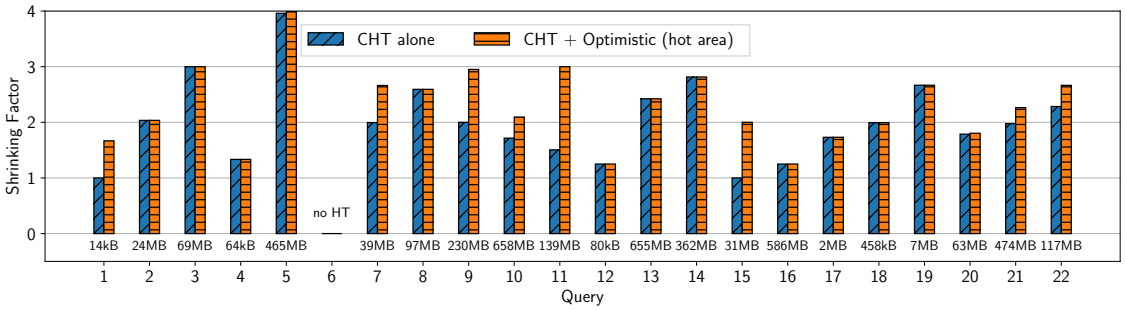


Figure 3: Reduction in hash table memory footprint over TPC-H with baseline hash table memory footprint (below the bar)

pen because (a) the string is rejected based on our sampling strategy or (b) a probing sequence of longer than 3 in the linear hash table is detected (due to the low load factor, this is highly infrequent, yet keeps negative lookups fast).

Our sampling strategy gives priority to *string constants* that occur in the query text; these are inserted first. After that, scans will insert strings until the USSR is full. We argue that the fact that a string column is dictionary-compressed, indicates that strings stem from a domain with a small cardinality. Therefore, these strings are good candidates for insertion into the USSR.

Vectorwise stores and buffers data in compressed form and decompresses column slices on the fly in the table scan operator. When reading a new dictionary-compressed block, the scan needs to set up an in-memory array with string pointers. Strings are represented as pointers in-flight in a query and decompression means looking up dictionary codes into this array. Rather than pointing into the dictionary inside the buffered block, when setting up this array, the scan inserts all dictionary strings into the USSR, so (most of) these pointers will point into the USSR instead. Insertion may fail, in which case the pointers still point into the block.

The sampling strategy further tries to optimize usage of the limited data region, by failing inserts of long strings that occupy $> \min(F, \max(2, \lfloor \frac{F}{64} \rfloor))$ 8-byte slots, where F is the free space in the data region (in slots). The idea is that it is better to accept more small strings than a few large strings, in case space fills up.

4.5 Accelerating Hashing & Comparisons

The USSR can be used to speed up hash computations. After testing whether a given string resides in the USSR using a bit-wise `and` operation, one can directly access the pre-computed hash value, which physically precedes the string:

```
inline uint64_t hash(char* s) {
  if (((uintptr_t)s & USSR_MASK) != ussr_prefix)
    return strhash(s); // compute hash
  return ((uint64_t*) (s))[-1]; // exploit pre-computed hash
}
```

The USSR also speeds up string comparisons when both compared strings reside in it. We exploit the fact that all strings within the USSR are unique. Hence, if the pointers are equal, the strings themselves are:

```
inline bool equal(char* s, char* t) {
  if ((((uintptr_t)s & USSR_MASK) != ussr_prefix) |
       (((uintptr_t)t & USSR_MASK) != ussr_prefix))
    return strcmp(s, t)==0; // regular string comparison
  return s==t; // in the USSR pointer equality is enough
}
```

4.6 Optimistic Splitting & the USSR

Optimistic Splitting and the USSR are complementary. The idea is to store USSR-backed strings, as small integers, compactly in the hot area and heap-backed strings in the cold area. Specifically, rather than storing string pointers in the hot area, we store slot numbers, pointing into the USSR. As mentioned earlier, these slot numbers are limited to 2^{16}, so they can be represented as unsigned 16-bit integers.

During packing, we represent exceptions using the invalid slot number 0 in the hot area of the hash table, and store the full 64-bit pointer in the exception area. Whenever a string needs to be unpacked, we first access the hot area and unpack the slot number. For non-zero slot numbers we can directly reconstruct the pointer of the string (`base address of USSR data region + slot*8`). However, we can further accelerate equality comparisons on strings by first comparing the slot numbers and, only if they are 0, comparing the full strings. A USSR encoded string `p` can be translated into a slot number quickly using `(p >> 3) & 65535`.

5. EXPERIMENTAL EVALUATION

In this section, we provide an experimental evaluation of our contributions to show that our techniques improve performance as well as memory footprint.

For this evaluation, we integrated Domain-Guided Prefix Suppression, Optimistic Splitting, and the USSR into Vectorwise. Besides generating all necessary function kernels, we had to extend the domain derivation mechanism and implement our greedy packing algorithm. In addition, we modified the existing hash table implementation, extended the hash join operator to take advantage of compressed key and payload columns, as well as the hash aggregation (group-by) operator to support Optimistic Aggregates.

We first evaluate the end-to-end performance on the TPC-H benchmark. We then present a high-level comparison on a real BI workload Afterwards we move to micro-benchmarks, analyze and discuss the impact of the USSR on string-intensive queries. Then we evaluate the hash probe performance over varying hash table sizes and the influence of different domains on hash table performance.

All experiments were performed on a dual-socket Intel Xeon Gold 6126 with 12 physical cores, 19.25 MB L3 cache each and is equipped with 384 GB of main memory. All results stem from hot runs using single-threaded execution.

5.1 TPC-H Benchmark

We evaluated the impact of Domain-Guided Prefix Suppression, Optimistic Splitting and the USSR on the widely used TPC-H benchmark with scale factor 100. We executed

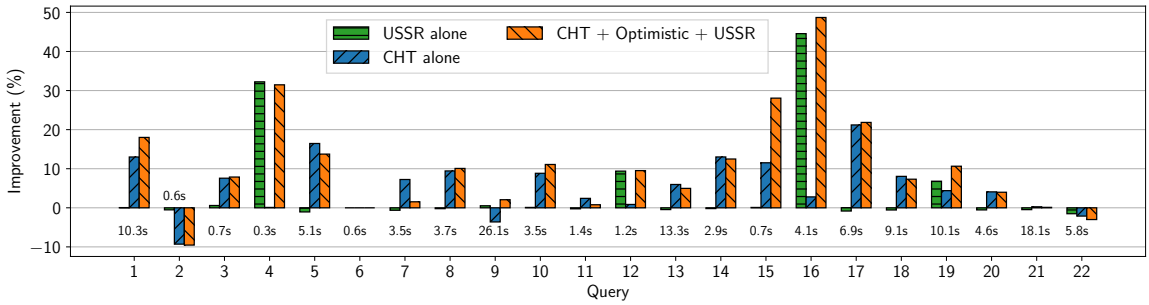


Figure 4: Improvement over TPC-H power run with baseline times (under the bar)

all 22 queries on our modified Vectorwise with and without our optimizations. We measured hash table memory footprint, as well as query response time.

Memory Footprint: In Vectorwise the memory consumption of many queries, particularly the TPC-H queries, is dominated by the size of hash tables. Therefore, during the TPC-H power run, we measured hash table sizes. Figure 3 shows the compression ratios we measured.

Domain-Guided Prefix Suppression (*CHT alone*), without Optimistic Splitting and USSR, was able to reduce hash table size by up to 4×. However, due to certain hurdles the compression ratio is often limited to 2×:

(a) Aggregates are not compressible without Optimistic Splitting. (b) Without the USSR, each string has to be a 64-bit pointer into a string heap. On recent hardware, this requires storing at least 48 bits with Domain-Guided Prefix Suppression. (c) As CHT does not make sense for CPU cache-resident hash tables, we do not enable it if the hash table is small, based on optimizer estimates. The impact of (a) and (b) on the active working set will be reduced using Optimistic Splitting and the USSR.

Optimistic Splitting aims at improving performance through more efficient cache utilization by separating the hash table into a thin frequently-accessed table (hot area) and a rarely accessed table (cold area). In combination with the USSR we measured a 2–4× smaller hot area (*CHT + Optimistic (hot area)*) in many TPC-H queries.

However, Optimistic Splitting in fact *increases* (rather than reduces) the memory consumption as it introduces additional data. For example, splitting a 128-bit `SUM` aggregate will introduce an additional aggregate with a smaller size but the full 128-bit aggregate will still reside in cold area.

Query Performance: To demonstrate the performance benefits of the USSR, Domain-Guided Prefix Suppression and Optimistic Splitting, we visualize the query response times of all 22 TPC-H queries in Figure 4. We split our analysis into three stages. First, we evaluate the impact achieved by only using the USSR. Then we discuss the effects of only using Domain-Guided Prefix Suppression. Finally, we discuss the influence of the combination of all three techniques.

The idea of the USSR is to boost operations on frequent strings. However, TPC-H is not an extremely string-intensive benchmark. Nonetheless, by using the Unique Strings Self-aligned Region (*USSR alone*) three queries (Q4, Q12 and Q16) showed significant performance gains. All three benefit from faster string hashing and equality comparisons provided by the USSR and improve by up to 45%.

Apart from the string-specific USSR, Domain-Guided Prefix Suppression aims at shrinking hash tables and providing operations on compressed data. Domain-Guided Prefix Suppression accelerates most queries (*CHT alone*) by up to 30%. In most queries we noticed an improvement of at least 10%. This is caused by the more efficient expression evaluation that smaller data types provide and the more cache-efficient hash table that allows equality comparisons directly on compressed keys. Notably, the regression in Q2 was caused by type casting overhead due to opportunistic shrinking of data types. We highlight that the purpose of Domain-Guided Prefix Suppression is mostly to reduce the memory footprint and not necessarily to speedup query evaluation.

When combining all three techniques (Domain-Guided Prefix Suppression, USSR and Optimistic Splitting) we measured gains up to 40% (*CHT + Optimistic + USSR*). We measured additional improvements from 5%, in Q1, up to 10%, in Q15. Both queries benefited from the Optimistic `SUM` aggregate which boosted the aggregate computation.

5.2 Public BI Benchmark

It has been noted that synthetic benchmarks like TPC-H do not capture all relevant aspects of real workloads [7, 10]. Recently, a workload study was published [24] based on the Tableau Public [1] Business Intelligence (BI) free cloud service. It analyzes its workbooks (data and queries generated by the Tableau BI tool) and specifically notes that users make extensive use of string data types (i.e. strings are by far the most common data type; used for 49% of all values). Not only is text data prevalent in these workbooks, but it is also observed that date columns, numeric and decimal columns are often stored as strings; arguably sub-optimally, but often this is related to data cleaning issues. Regrettably, this study did not publish the data and queries as an open benchmark, also upon our request to Tableau. Inspired by this work, we manually downloaded the 48 biggest Tableau Public workbooks (400 GB data) and extracted the SQL statements from its query log. This workload is now available in open-source as the *Public BI Benchmark* [2]. As a representative example, we focus on one of its workbooks:

CommonGovernment. We extracted all 43 queries and all 13 tables. Each table contains around 8 GiB of data in CSV format. Unlike TPC-H, each table contains many string columns and columns that contain `NULL` values are common. We executed each query sequentially and Table 2 shows the measured effects on the runtime.

The workbook *CommonGovernment* is string-intensive: using only the USSR, we measured a speedup of up to ≈ 2× (55% improvement). These speedups are caused by (a) many strings residing in the USSR, because they originate from a small domain of unique strings, and (b) many strings

Table 2: Speedup and USSR statistics for workbook *CommonGovernment*

Query	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Speedup	**2.1**	**1.4**	**2.2**	**1.4**	**1.3**	**1.0**	**1.2**	**1.0**	**1.5**	**1.8**	**1.1**	**2.2**	**2.2**	**1.8**	**1.5**	**2.1**	**1.4**	**1.1**	**1.4**	**1.1**
USSR Size (kB)	1.8	0.5	2.0	0.3	66.1	512.0	83.2	512.0	12.7	7.2	112.4	1.9	1.8	7.2	1.8	2.0	1.8	110.3	0.3	512.0
Rejection Ratio (%)	0.0	0.0	0.0	0.0	0.0	18.3	0.0	32.8	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	21.1
#Rejected	0	0	0	0	0	37627	0	30204	0	0	0	0	0	0	0	0	0	17	0	13742
#Candidates	1312	2720	1056	1504	73200	205440	77296	92208	656	6768	110704	1072	1232	6752	3792	1360	3808	99744	1728	65222
#Strings in USSR	46	16	49	11	2251	12227	2835	12218	252	165	3041	48	45	167	51	49	51	2990	11	21343
Average String Length	23	3	20	5	18	26	19	29	21	25	23	22	22	25	18	22	19	23	5	11
Baseline Runtime (s)	0.15	0.27	0.13	0.17	0.37	3.48	0.39	0.54	0.18	0.16	0.29	0.14	0.14	0.16	0.18	0.14	0.17	0.27	0.18	0.51
Baseline HT size (MB)	0.05	0.09	0.05	0.09	0.14	82.11	0.14	9.12	0.07	0.05	0.11	0.05	0.05	0.05	0.05	0.05	0.05	0.11	0.09	8.11

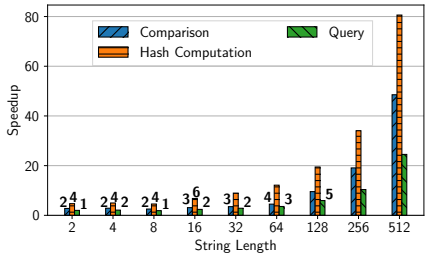


Figure 5: Group-By on string keys: Speedup vs. length

are long enough to significantly impact string operations to cause a speedup of the whole query.

Q6, Q8 and Q20 show no significant benefits from the USSR, mainly because the string columns have a large unified dictionary (that does not even fit fully in the USSR). While dictionary-coded decompression in Vectorwise has a sub-cycle per/tuple cost, the effort of setting up the dictionary array when the scan moves to a new disk block increases, when the per-block dictionary size increases. With the USSR, this setting-up effort becomes significantly higher as all dictionary strings must be looked up in the USSR linear hash table. Also, with larger dictionaries per block, each dictionary string has a lower repetition count during execution; so the amortization of the setting-up investment by faster hashing and comparison decreases. Still, we see that we make a good trade-off, as queries Q8 and Q20 still get (marginally) faster, and only Q6 is marginally slower.

In general, the Public BI workload is characterized by few joins and many aggregations [24], where these aggregations produce small results—few or in the thousands, but almost never in the millions of tuples. This means that the hash tables needed for aggregation are often CPU cache-resident. Therefore, CHT is not triggered and that the USSR is what most matters in this workload, so we focus only on that.

5.3 Micro-Bench: USSR and Group-By

We now move to a number of micro-benchmarks to focus on individual performance aspects of string processing with the USSR. We start with the performance on a `SELECT COUNT(*) FROM T GROUP BY s` query. These strings came from a domain of 10 unique strings, all strings had the same length. Figure 5 shows the speedups that can be achieved using the USSR. We profiled the time spent on string comparisons when checking the keys inside group by's hash table. This results show significant speedups reaching from a 2× to 50× faster string comparison. Similarly, we profiled the time spent on computing hash of the string keys. This results in speedups reaching from 4× for small strings, to 80× for large strings. Besides the significant speedup in terms of string comparison and hash computation, we also noticed significant speedup of the whole query up to ≈ 25×.

5.4 Micro-Bench: Join Probe Performance

We now micro-benchmark Domain-Guided Prefix Suppression, with respect to hash table lookup performance. Our experiment consists of a simple join query where we vary the size of the inner/build relation and the domains of the key columns. We experimented with two and four key columns, four payload columns with values $v \in [0, 10]$.

Figure 6 visualizes the speedup, as well as, the L3/last-level cache (LLC) misses measured. We observe an up to 2.5× faster hash probe including the tuple reconstruction cost. The measured speedups tend to increase with hash table size (size of inner relation). For large hash tables with more than 10^6 rows, the speedups were caused by the significantly smaller and, consequently, more cache-resident hash table. For hash tables with less than 10^6 rows, the performance was mostly affected by the more efficient comparisons directly on compressed data.

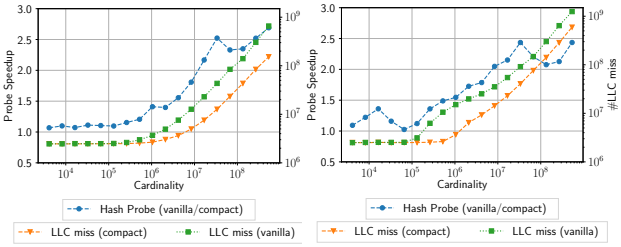
6. RELATED WORK

In this paper we aim at improving cache efficiency of hash tables through compression. An alternative approach is to increase the fill-rate using techniques such as Robin Hood Hashing [9] and Cuckoo Hashing [19]. Similarly, Concise Hash Tables [5] are optimized linear hash tables which try to omit the storage of empty rows. A major disadvantage of Concise Hash Tables is the restriction to linear hash tables, whereas our techniques can be applied to linear as well as bucket-chained hash tables. An extension of Concise Hash Tables are Concise Array Tables [5] which avoid storing the keys by providing a collision-free hash table. These four techniques are orthogonal to our approach of treating the hash table as a compressed table. We highlight that these approaches can be applied, in addition to our techniques, to achieve even more cache-efficient hash tables.

The use of compression is widely spread among database systems. Commonly, analytic databases utilize lightweight compression schemes [26] to elevate effective memory and disk bandwidth during table scans. Although they used similar techniques as ours, notably bit-packing, frame-of-reference and dictionary compression, in this work, we apply compression to hash tables, data structures used *during query evaluation* rather than in data storage. An application of compression which has not received much attention.

The possibility of exploiting compressed data inside data processing pipelines, i.e. compressed execution, is investigated by Abadi et al. [3]. Evaluating predicates on compressed data has been explored deeply for table scans [12, 14, 17, 25]. However, these works focus on scans in isolation.

Lee et al. [16] pioneered joins on data encoded using per-column on-the-fly dictionaries. These dictionaries might seem closely related to the USSR. However there are two major differences: (1) The use of multiple dictionaries necessitates a translation from one encoded join column to the



(a) 4 keys $k_1, ..., k_4 \in [0, 1.000]$ whereas schema suggests 64-bit integers

(b) 2 keys $k_1, k_2 \in [0, 10^6]$ whereas schema suggests 128-bit integers

Figure 6: Hash probe speedup vs. build-side cardinality using 4 payload columns $p_1, ..., p_4 \in [0, 10]$

encoding of the other. The USSR, being a unified query-wide dictionary, does not require such a translation. (2) To fit into cache the USSR has a small fixed size, as opposed to the on-the-fly dictionaries that can grow very large.

Many database systems allow evaluating simple predicates directly on compressed data. Notable examples are IBM BLU [21], SQLServer [15], Quickstep [20], and HyPer [14]. But only very few systems exploit compressed data inside query pipelines. Most notably IBM BLU [21] supports operations on compressed data. It performs joins on encoded and partitioned data similar to Lee et al. [16] and, hence, suffers from the same disadvantages.

Shatdal et al. [23] proposed to optimize algorithms for cache efficiency. One of their techniques, key extraction in sorting/partitioning shares some similarity with Optimistic Splitting. However, Optimistic Aggregates are continuously updated, whereas extracted keys stay constant.

Encoding strings inside dictionaries has been explored by Färber et al. [11] and Binnig et al. [6]. However, both assume global dictionaries, which we do not require (Section 4.1). In contrast, the USSR is a small query-wide on-the-fly dictionary which only encodes frequent strings and avoids expensive update and delete operations.

7. SUMMARY

Hash tables are crucial data structures for modern query engines. However, in analytical queries, they consume a significant amount of memory. Shrinking hash tables not only lowers the memory footprint, but also leads to faster access by fitting more data into faster memory. We present three composable techniques to shrink hash tables: Domain-Guided Prefix Suppression (extremely lightweight compression), Optimistic Splitting (decomposition into hot and cold value slices), and the USSR (opportunistic dictionary compression). We implemented these techniques in the industrial-strength DBMS Vectorwise. In our experiments, our techniques improved query performance by up to 25× in string-intensive queries. On the synthetic TPC-H benchmark we noticed up to 4× smaller hash tables and improved query runtime by up to 50%. On the realistic Public BI workload we achieved improvements of up to 2.2×.

8. REFERENCES

[1] `https://public.tableau.com`.

[2] `https://github.com/cwida/public_bi_benchmark`.

[3] D. Abadi, S. Madden, and M. Ferreira. Integrating compression and execution in column-oriented database systems. In *SIGMOD*, pages 671–682, 2006.

[4] C. Balkesen, J. Teubner, G. Alonso, and M. T. Özsu. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *ICDE*, pages 362–373, 2013.

[5] R. Barber, G. Lohman, I. Pandis, V. Raman, R. Sidle, G. Attaluri, N. Chainani, S. Lightstone, and D. Sharpe. Memory-efficient hash joins. *PVLDB*, 8(4):353–364, 2014.

[6] C. Binnig, S. Hildenbrand, and F. Färber. Dictionary-based order-preserving string compression for main memory column stores. In *SIGMOD*, pages 283–296, 2009.

[7] P. Boncz, A.-C. Anatiotis, and S. Kläbe. JCC-H: Adding join crossing correlations with skew to TPC-H. In *TPCTC*, pages 103–119, 2017.

[8] P. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-pipelining query execution. In *CIDR*, pages 225–237, 2005.

[9] P. Celis. *Robin Hood Hashing*. PhD thesis, University of Waterloo, 1986.

[10] A. Crolotte and A. Ghazal. Introducing skew into the TPC-H benchmark. In *TPCTC*, pages 137–145, 2012.

[11] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner. SAP HANA database: Data management for modern business applications. *SIGMOD Record*, pages 45–51, 2012.

[12] B. Hentschel, M. S. Kester, and S. Idreos. Column Sketches: A scan accelerator for rapid and robust predicate evaluation. In *SIGMOD*, pages 857–872, 2018.

[13] S. Jain, D. Moritz, D. Halperin, B. Howe, and E. Lazowska. SQLShare: Results from a multi-year SQL-as-a-service experiment. In *SIGMOD*, pages 281–293, 2016.

[14] H. Lang, T. Mühlbauer, F. Funke, P. Boncz, T. Neumann, and A. Kemper. Data Blocks: Hybrid OLTP and OLAP on compressed storage using both vectorization and compilation. In *SIGMOD*, 2016.

[15] P. Larson, C. Clinciu, C. Fraser, E. N. Hanson, M. Mokhtar, M. Nowakiewicz, V. Papadimos, S. L. Price, S. Rangarajan, R. Rusanu, and M. Saubhasik. Enhancements to SQL server column stores. In *SIGMOD*, pages 1159–1168, 2013.

[16] J.-G. Lee, G. Attaluri, R. Barber, N. Chainani, O. Draese, F. Ho, S. Idreos, M.-S. Kim, S. Lightstone, G. Lohman, K. Morfonios, K. Murthy, I. Pandis, L. Qiao, V. Raman, V. K. Samy, R. Sidle, K. Stolze, and L. Zhang. Joins on encoded and partitioned data. *PVLDB*, 7(13):1355–1366, 2014.

[17] Y. Li and J. Patel. BitWeaving: Fast scans for main memory data processing. In *SIGMOD*, pages 289–300, 2013.

[18] I. Müller, C. Ratsch, and F. Färber. Adaptive string dictionary compression in in-memory column-store database systems. In *EDBT*, pages 283–294, 2014.

[19] R. Pagh and F. Rodler. Cuckoo hashing. *J. Algorithms*, 51(2):122–144, May 2004.

[20] J. Patel, H. Deshmukh, J. Zhu, N. Potti, Z. Zhang, M. Spehlmann, H. Memisoglu, and S. Saurabh. Quickstep: A data platform based on the scaling-up approach. *PVLDB*, 11(6):663–676, 2018.

[21] V. Raman, G. Attaluri, R. Barber, N. Chainani, D. Kalmuk, V. Kulandaisamy, J. Leenstra, S. Lightstone, S. Liu, G. Lohman, T. Malkemus, R. Mueller, I. Pandis, B. Schiefer, D. Sharpe, R. Sidle, A. Storm, and L. Zhang. DB2 with BLU acceleration: So much more than just a column store. *PVLDB*, 6(11):1080–1091, 2013.

[22] S. Schuh, X. Chen, and J. Dittrich. An experimental comparison of thirteen relational equi-joins in main memory. In *SIGMOD*, pages 1961–1976, 2016.

[23] A. Shatdal, C. Kant, and J. F. Naughton. Cache conscious algorithms for relational query processing. VLDB, pages 510–521, 1994.

[24] A. Vogelsgesang, M. Haubenschild, J. Finis, A. Kemper, V. Leis, T. Muehlbauer, T. Neumann, and M. Then. Get real: How benchmarks fail to represent the real world. In *DBTEST*, 2018.

[25] T. Willhalm, N. Popovici, Y. Boshmaf, H. Plattner, A. Zeier, and J. Schaffner. SIMD-scan: Ultra fast in-memory table scan using on-chip vector processing units. *PVLDB*, 2(1):385–394, 2009.

[26] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-scalar RAM-CPU cache compression. In *ICDE*, 2006.

[27] M. Zukowski, N. Nes, and P. Boncz. DSM vs. NSM: CPU performance tradeoffs in block-oriented query processing. In *DaMoN*, pages 47–54, 2008.

Technical Perspective: Probabilistic Data with Continuous Distributions

Dan Olteanu
Department of Informatics, University of Zurich
dan.olteanu@uzh.ch

The paper entitled "Probabilistic Data with Continuous Distributions" overviews recent work on the foundations of *infinite* probabilistic databases [3, 2]. Prior work on probabilistic databases (PDBs) focused almost exclusively on the *finite* case: A finite PDB represents a discrete probability distribution over a finite set of possible worlds [4]. In contrast, an infinite PDB models a continuous probability distribution over an infinite set of possible worlds. In both cases, each world is a finite relational database instance. Continuous distributions are essential and commonplace tools for reasoning under uncertainty in practice. Accommodating them in the framework of probabilistic databases brings us closer to applications that naturally rely on both continuous distributions and relational databases.

The infinite sample space of possible worlds raises significant technical challenges. I was therefore excited to learn from this work how to address the foundational questions of the representation and querying of probabilistic data in an elegant and technically sound way:

- how to represent succinctly an infinite set of possible worlds in finite space;
- how to get the right semantics for queries over infinite PDBs that natural extends the finite case.

These questions require a deep mix of mathematics and database theory, which can be easily intimidating. This overview paper is to be commended for its approachable style focused on explaining the challenges one at a time and how they were overcome.

Representation. The first question is to effectively represent continuous distributions over an infinite set of possible worlds. The overview paper shows how the formalisms of tuple-independent PDBs [4], block-independent disjoint PDBs [4], and Generative Datalog [1] can be adapted to the infinite setting.

One major difficulty when defining probabilities on uncountable spaces is that we cannot assign a well-defined probability to all subsets of the space, but only to subsets that are *measurable*. Topological Polish spaces and σ-algebras are used to define the event space in a sufficiently generic way, which only depends on the database schema and the domains of the attributes in the schema, while making sure that all sets of interest are measurable, such as those defined by views over PDBs.

Querying. The second question is what would be a well-defined semantics for views that map between infinite PDBs. It turns out that such a semantics is subject to measurability, now lifted to view mappings with respect to the σ-algebras of input and output PDBs. A remarkable result is that the views expressible in relational calculus with aggregation and in Datalog (including variants such as Inflationary Datalog and Least Fixed-Point Logic) are measurable and therefore admit a well-defined semantics over infinite PDBs [3].

The paper also gives a sound semantics to Generative Datalog with continuous distributions that naturally generalises the known case of discrete distributions [2]. Generative Datalog is a declarative probabilistic programming language for relational data, where the Datalog rules may specify distributions at the place of query variables in the head [1]. Deterministic relational data is fed into a generative model that defines a probability distribution over possible database instances. Whenever the rule body is satisfied, the head fires and a new sample is generated for the rule head. In case of continuous distributions, repeatedly sampling under the same satisfaction of the rule body may lead to non-termination. A further challenge is ensuring the order of rule applications is immaterial, which is essential for language declarativeness.

The work overviewed in the paper mirrors development on the foundations of finite PDBs: It investigates representation formalisms and their closure under various query languages. This is a prerequisite for studies on: the tractability of query evaluation on infinite PDBs; exact and approximate query evaluation algorithms; and systems for managing relational data with continuous probability distributions.

1. REFERENCES

[1] V. Bárány, B. ten Cate, B. Kimelfeld, D. Olteanu, and Z. Vagena. Declarative probabilistic programming with datalog. *TODS*, 42(4):22:1–22:35, 2017.
[2] M. Grohe, B. L. Kaminski, J. Katoen, and P. Lindner. Generative datalog with continuous distributions. In *PODS*, pages 347–360, 2020.
[3] M. Grohe and P. Lindner. Infinite probabilistic databases. In *ICDT*, pages 16:1–16:20, 2020.
[4] D. Suciu, D. Olteanu, C. Ré, and C. Koch. *Probabilistic Databases*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2011.

Probabilistic Data with Continuous Distributions*

Martin Grohe
RWTH Aachen University
Germany
grohe@informatik.rwth-aachen.de

Benjamin Lucien Kaminski
University College London
United Kingdom
b.kaminski@ucl.ac.uk

Joost-Pieter Katoen
RWTH Aachen University
Germany
katoen@informatik.rwth-aachen.de

Peter Lindner
RWTH Aachen University
Germany
lindner@informatik.rwth-aachen.de

ABSTRACT

Statistical models of real world data typically involve continuous probability distributions such as normal, Laplace, or exponential distributions. Such distributions are supported by many probabilistic modelling formalisms, including probabilistic database systems. Yet, the traditional theoretical framework of probabilistic databases focuses entirely on finite probabilistic databases.

Only recently, we set out to develop the mathematical theory of infinite probabilistic databases. The present paper is an exposition of two recent papers which are cornerstones of this theory. In (Grohe, Lindner; ICDT 2020) we propose a very general framework for probabilistic databases, possibly involving continuous probability distributions, and show that queries have a well-defined semantics in this framework. In (Grohe, Kaminski, Katoen, Lindner; PODS 2020) we extend the declarative probabilistic programming language Generative Datalog, proposed by (Bárány et al. 2017) for discrete probability distributions, to continuous probability distributions and show that such programs yield generative models of continuous probabilistic databases.

1. INTRODUCTION

Probabilistic databases [20, 21, 22] provide a framework for quantitatively modelling uncertainty in data. Sources of uncertainty are numerous; common examples are noisy sensor data, data gathered from unreliable sources, and inconsistent data. Formally, a probabilistic database (PDB) is a probability space over database instances, called the possible worlds. Traditionally, these probability spaces were limited to be finite. This implies a closed world assumption where only finitely many facts could possibly be true, and it rules out any probability distributions with an infinite support. Yet, in many applications, infinitely, even uncountably infinitely, supported probability distributions arise naturally, and many real-world statistical phenomena follow infinite probability distributions such as Poisson distributions, normal distributions, or exponential distributions.

Example 1. Suppose we have a relation storing room temperatures, as in Figure 1(a). The temperature measurements may be noisy, and we may account for this by adding a normally distributed error with a small variance $\varepsilon > 0$, resulting in a simple probabilistic relation which may be represented as in Figure 1(b). ⌟

Example 2. In this example, consider a particle detector such as the Alpha Magnetic Spectrometer (AMS-02[1]) on the ISS. Suppose we record the detected particles in a relation of schema (`Time`, `Trajectory`, `Velocity`). As in the previous example, the measurements (of the trajectory and velocity) may be imprecise and best modelled by a probability distribution. But here we have an additional source of uncertainty: some particles may go undetected. If we also model this type of error, the number of tuples in the relation becomes a random variable as well. Then there is no a-priori bound on the size of the instances in the resulting PDB. Note, however, that every instance is still finite, because in every time interval only finitely many particles can hit the detector, and our model should account for that. ⌟

Both examples exhibit probabilistic databases with continuous probability distributions that cannot be captured by the traditional model of finite probabilistic databases. Generalising from finite to continuous probability distributions comes with a substantial mathematical overhead. While PDBs of fixed (or bounded) size, such as those arising from Example 1, are still relatively easy to handle, PDBs of unbounded size such as the one we saw in Example 2 are nontrivial to capture mathematically, let alone to deal with algorithmically. Several PDB systems that have been proposed over the years [2, 13, 14, 19] handle continuous probability distributions. The flexibility of these systems reaches as far as providing declarative representations of continuous probabilistic databases and even continuous-space database-valued Markov processes. Yet, only recently [10, 12], we

*The results presented in this paper were originally published in M. Grohe and P. Lindner: *Infinite Probabilistic Databases*, Proc. ICDT 2020 and M. Grohe, B.L. Kaminski, J.-P. Katoen, P. Lindner: *Generative Datalog with Continuous Distributions*, Proc. PODS 2020.

[1]See `https://ams02.space/`.

RoomNo	Time	°C
4108	2021-01-05 08:00	20.2
4108	2021-01-05 14:00	21.8
4109	2021-01-05 08:00	22.1
⋮	⋮	⋮

(a)

RoomNo	Time	°C
4108	2021-01-05 08:00	$\mathcal{N}(20.2, \varepsilon)$
4108	2021-01-05 14:00	$\mathcal{N}(21.8, \varepsilon)$
4109	2021-01-05 08:00	$\mathcal{N}(22.1, \varepsilon)$
⋮	⋮	⋮

(b)

Figure 1: A relation storing room temperatures: (a) deterministically and (b) with a normally distributed error

proposed a general framework for rigorously dealing with probabilistic databases over continuous domains and provided a sound semantics for standard query languages such as the relational calculus. We will present this framework in Sections 4 and 5 of this paper. To distinguish them from the traditional "finite" PDBs, we call PDBs with an infinite sample space *infinite PDBs* in the following. Note that every instance in an infinite PDB is just a standard finite relational database instance, it is only the sample space of all possible instances that is infinite.

A difficult issue when dealing with PDBs is how to efficiently represent them. This problem already arises for finite PDBs, but is much more fundamental when dealing with infinite PDBs that do not even allow for a naive representation that explicitly lists all instances. So we have to rely on *implicit* representations, which can either be ad-hoc representations such as the one chosen to illustrate Example 1 in Figure 1(b) or generic formalisms for representing complex probability distributions, such as probabilistic graphical models, deep neural networks, and probabilistic programming languages. Yet, when dealing with (relational) PDBs, it is desirable to stay within the declarative framework of relational databases. To this end, Bárány, ten Cate, Kimelfeld, Olteanu, and Vagena [3] introduced a declarative probabilistic programming language based on Datalog, which has a generative part enabling to represent complex probability distributions strictly within the framework of relational databases. However, the semantics of Bárány et al. is only able to handle discrete probability distributions. In [9], we generalised the semantics to continuous distributions. The resulting *Generative Datalog* can serve both as a powerful representation language for relational PDBs with discrete and continuous distributions and as a query language for PDBs. We present this language in Section 7.

The reader may wonder if it is really necessary to consider continuous probabilistic databases. After all, they can only be mathematical abstractions of real systems, where instead of the continuum of real numbers we only see the finite set of 64 bit floating point numbers. Then aren't finite probabilistic databases all we need? Well, the history of computer science has shown us that the right abstractions can be extremely powerful—just think of the relational database model— and certainly we do not want the semantics of our query languages depend on whether we use 32 or 64 bit floating-point numbers to specify probabilities. All of applied mathematics, including statistics, uses the real numbers as the right abstraction to reason about continuous phenomena. And when reasoning about uncertain and probabilistic data, we want to have standard tools such as normal distributions at our disposal.

2. TOWARDS INFINITE PDBS

Before we delve into the mathematical details, in this section we describe the general approach on an intuitive level and highlight the technical difficulties we are facing.

We define a probabilistic database to be a probability space whose sample space consists of database instances of some schema τ. In the traditional approach, this probability space is assumed to be finite; here, we would like to allow it to be infinite. The difficulty when defining probabilities on uncountable spaces such as the reals is that we cannot assign a well-defined probability to all subsets of the space, but only to subsets that are *measurable*.

Let us ignore this issue for a moment (though it will come back to bite us) and think about *how we can actually define a probability distribution on uncountable sets of database instances*. Let us fix a simple database schema τ consisting of a single binary relation R of schema (`Time`, `Value`), where the attribute `Value` is real-valued. Instances are relations of this schema. We can also view them as finite sets (without duplicates) or finite bags (possibly with duplicates)—depending on the type of semantics we are interested in—of facts of the form $R(t, v)$, where t is a point in time and v a real number. If we want to define a finite probability space on the instances, we can simply pick a finite set $D_1, \ldots, D_m$ of instances and assign probabilities $p_1, \ldots, p_m \in [0, 1]$ to them such that $\sum_i p_i = 1$. We can extend this approach to countably infinite spaces, but not to uncountable spaces, where typically every single instance has probability 0. This happens, for example, if we assume the `Value` to be normally distributed at any `Time`. We know how to define a probability distribution on the `Value`s (that is, the real numbers); we only need to specify the probability mass on each interval. But here we need to define a probability distribution on *sets* or *bags* of `Time`-`Value` pairs. It is not at all obvious how to do that, except maybe in simple settings such as the one described in Example 1. We need to draw from the theory of *finite point processes* [17, 16, 7]. In probability theory, point processes are used to describe probability spaces of finite or countable sets or bags. Based on the theory of point processes, we will define a very general framework for infinite PDBs that we call *standard PDBs* (see Section 4).

Once we have defined our probability spaces, we need to think about *querying PDBs*. To define the semantics of queries and views, let us consider a view V mapping instances of schema τ to instances of schema τ'. Queries are just specific views where the target schema τ' consist of a single relation schema. We want to define a semantics for this view V on probabilistic databases, that is, we want to extend it to a mapping from PDBs of schema τ to PDBs of schema τ'. Let us assume that we have a PDB $\mathcal{D}$ of schema τ, and we want to define the image $V(\mathcal{D})$, which is supposed to be a PDB of schema τ'. To do this, for a set $\boldsymbol{D}'$ of instances of schema τ' we define the probability of $\boldsymbol{D}'$

RoomNo	Time	°C
4108	2021-01-05 08:00	$\mathcal{N}(20.2, 0.1)$
4108	2021-01-05 14:00	$\mathcal{N}(21.8, 0.1)$
4109	2021-01-05 08:00	$\mathcal{N}(22.1, 0.1)$
4109	2021-01-05 14:00	$\mathcal{N}(22.3, 0.1)$
4109	2021-01-06 08:00	$\mathcal{N}(21.9, 0.1)$

Figure 2: A PDB of schema $\tau = \{\texttt{Temp}(\texttt{RoomNo}, \texttt{Time}, \texttt{°C})\}$

in $V(\mathcal{D})$ to be the probability of the set $V^{-1}(\boldsymbol{D}')$ in $\mathcal{D}$:

$$\Pr_{V(\mathcal{D})}(\boldsymbol{D}') := \Pr_{\mathcal{D}}\big(V^{-1}(\boldsymbol{D}')\big). \tag{A}$$

Example 3. Recall Example 1, where we considered PDBs of a schema $\tau = \{\texttt{Temp}(\texttt{RoomNo}, \texttt{Time}, \texttt{°C})\}$. Entries are room temperatures at various times. Let Q be the query that maps instances of schema τ to instances of schema $\tau' = \{\texttt{AvTemp}(\texttt{RoomNo}, \texttt{°C})\}$ recording the average temperature in each room, defined by the SQL-expression

```
SELECT RoomNo, AVG(°C) FROM Temp GROUP BY RoomNo.
```

Let us apply this query to the PDB $\mathcal{D}$ represented by the relation `Temp` shown in Figure 2. Note that in all instances of this PDB, the table `Temp` has exactly five rows recording the temperatures in room 4108 at two different times and the temperatures in room 4109 at three different times. For simplicity, we assume that the random variables describing the entries in the five rows are independent.

In every instance of $Q(\mathcal{D})$, the table `AvTemp` has exactly two rows recording the average temperatures in rooms 4108 and 4109. We can easily compute the probabilities in $Q(\mathcal{D})$. For example, the probability that both rooms have an average temperature in the range 20–22 degrees equals the probability that the average of two normally distributed random variables with means $20.2, 21.8$ and variance 0.1 is between 20 and 22 times the probability that the average of three normally distributed random variables with means $22.1, 22.3, 21.9$ and variance 0.1 is between 20 and 22. Actually, the table `AvTemp` in $Q(\mathcal{D})$ can be represented as follows.

RoomNo	°C
4108	$\mathcal{N}(21.0, 0.05)$
4109	$\mathcal{N}(22.1, 0.33)$

The fact that a linear combination of normal distributions is again a normal distribution enables us to represent $Q(\mathcal{D})$ in such a simple "closed form". In general, views of PDBs can be far more complicated than the original PDBs. ⌟

Unfortunately, there is a subtle issue that we have neglected when defining the semantics of views and queries over PDBs. Recall that in uncountable probability spaces, we cannot define probabilities for all subsets of the sample space, but only for so-called *measurable* sets. This means that in the definition (A), we only need to consider measurable sets $\boldsymbol{D}'$ of instances of schema τ', but we need to make sure that the set $V^{-1}(\boldsymbol{D}')$ is measurable as well, for otherwise the probability on the right-hand side of (A) is not defined. This means that a view V only has a well-defined semantics on probabilistic databases if for every measurable set $\boldsymbol{D}'$ in the target space the pre-image $V^{-1}(\boldsymbol{D}')$ is a measurable set in the source space. If this is the case, we call V measurable. *Only measurable views and queries have a well-defined semantics on probabilistic databases.* Fortunately, it turns out that all views defined in standard query languages such as the relational calculus or Datalog are measurable. But this is a nontrivial result (Theorem 1). In [12, Example 8], we give an example of a relatively simple "query" that is not measurable.

3. MATHEMATICAL BACKGROUND

In this section, we collect some mathematical background underlying our approach to PDBs. The reader may skip this section and use it as a reference whenever needed later.

Topology

Topology qualitatively captures concepts such as closeness, convergence, and continuity, and it is the foundation for the measure theory and probability theory we need here. A *topology* on a set $\mathbb{X}$ is a family $\mathfrak{O}$ of subsets of $\mathbb{X}$ that contains $\mathbb{X}$ and the empty set and is closed under arbitrary unions and finite intersections. We call $(\mathbb{X}, \mathfrak{O})$ a *topological space* and the elements $O \in \mathfrak{O}$ *open sets*.

Example 4. (1) In the *standard topology* on the reals $\mathbb{R}$, a set $O \subseteq \mathbb{R}$ is open if for every $x \in O$ there is an $\varepsilon > 0$ such that $(x - \varepsilon, x + \varepsilon) \subseteq O$. Note that this topology is *generated* by the open intervals, which means that every open set is the union of open intervals.[2]

(2) For every set $\mathbb{X}$, the power set $2^{\mathbb{X}}$ is a topology on $\mathbb{X}$, the so-called *discrete topology*. ⌟

For $i = 1, 2$, let $(\mathbb{X}_i, \mathfrak{O}_i)$ be a topological space. A function $f\colon \mathbb{X}_1 \to \mathbb{X}_2$ is *continuous* (with respect to $\mathfrak{O}_1, \mathfrak{O}_2$) if $f^{-1}(O_2) \in \mathfrak{O}_1$ for every $O_2 \in \mathfrak{O}_2$.

Every *metric* d on $\mathbb{X}$ (that is, a distance function on pairs of elements of $\mathbb{X}$ that is symmetric, satisfies the triangle inequality, and has the property that two points have distance 0 if and only if they are equal) induces a topology on $\mathbb{X}$ where a set $O \subseteq \mathbb{X}$ is open if for every $x \in O$ there is an $\varepsilon > 0$ such that $\{y \mid d(x, y) < \varepsilon\} \subseteq O$. A topological space $(\mathbb{X}, \mathfrak{O})$ is *metrisable* if it is induced by a metric on $\mathbb{X}$ in this way. Obviously, the standard topology on the reals (Example 4(1)) is metrisable. The discrete topology on an arbitrary set $\mathbb{X}$ (Example 4(2)) is metrisable as well; as a metric we use the function d with $d(x, x) = 0$ and $d(x, y) = 1$ for all $x \neq y$, also know as the *discrete metric*.

A topological space $(\mathbb{X}, \mathfrak{O})$ is *separable* if there is a countable subset $Y \subseteq \mathbb{X}$ such that every nonempty open set $O \in \mathfrak{O} \setminus \{\emptyset\}$ contains an element from Y (we say that Y is *dense*). For example, for the reals with the standard topology, the set $\mathbb{Q}$ of rationals is a dense subset. The discrete topology on a set $\mathbb{X}$ is separable if and only if $\mathbb{X}$ is countable. Separability is a very important technical property in our arguments, because it enables us to work with countable approximations.

A final condition we need (though it is less important for us) is completeness: intuitively, a metrisable topological space $(\mathbb{X}, \mathfrak{O})$ is *complete* if every convergent sequence (more precisely, Cauchy sequence) converges to a point in $\mathbb{X}$. We omit the formal definition. A *Polish space* is a complete, separable, metrisable topological space (and its topology is *Polish*). The reals with the standard topology, all finite-dimensional Euclidean spaces, and all countable discrete topological spaces are Polish spaces.

[2]We take the union over the empty family of sets to be the empty set.

It is safe to say that all topological spaces we will ever find in database applications are Polish spaces.

Measure Theory and Probability

A *σ-algebra* on a set $\mathbb{X}$ is a set $\mathfrak{A}$ of subsets of $\mathbb{X}$ that contains the empty set and is closed under complementation and countable unions. A pair $(\mathbb{X}, \mathfrak{A})$, where $\mathfrak{A}$ is a σ-algebra on $\mathbb{X}$, is called a *measurable space*.

Example 5. (1) For every set $\mathbb{X}$, the set $\{\emptyset, \mathbb{X}\}$ and the power set $2^{\mathbb{X}}$ are σ-algebras on $\mathbb{X}$.
(2) Another σ-algebra on $\mathbb{X}$ is the set of all $Y \subseteq \mathbb{X}$ such that either Y is countable or $\mathbb{X} \setminus Y$ is countable.
(3) The set of all Lebesgue measurable subsets of the reals is a σ-algebra. ⌟

Let $\mathbb{X}$ be a set and $\mathfrak{S} \subseteq 2^{\mathbb{X}}$. The σ-algebra *generated* by $\mathfrak{S}$ is the closure of $\mathfrak{S}$ under complementation and countable intersections, that is, the smallest σ-algebra on $\mathbb{X}$ that contains $\mathfrak{S}$. Observe that the σ-algebra defined in Example 5(2) is the σ-algebra generated by all singleton sets $\{x\}$ for $x \in \mathbb{X}$.

For any topological space $(X, \mathfrak{O})$, the σ-algebra generated by the topology $\mathfrak{O}$ is called the *Borel σ-algebra* on $\mathbb{X}$, and its elements are called *Borel sets*. A measurable space $(\mathbb{X}, \mathfrak{A})$ is a *standard Borel space* if $\mathfrak{A}$ is the Borel σ-algebra of some Polish topology on $\mathbb{X}$. It is not difficult to show that if d is a metric inducing such a Polish topology and Y is a countable dense subset then $\mathfrak{A}$ is generated by the countable set of open balls $B_{1/n}(y) \coloneqq \{x \in \mathbb{X} \mid d(x, y) < 1/n\}$ for positive integers n and $y \in Y$. This is one of the reasons making standard Borel spaces very convenient to handle.

For $i = 1, 2$, let $(\mathbb{X}_i, \mathfrak{A}_i)$ be a measurable space. A function $f\colon \mathbb{X}_1 \to \mathbb{X}_2$ is *measurable* (with respect to $\mathfrak{A}_1, \mathfrak{A}_2$) if $f^{-1}(A_2) \in \mathfrak{A}_1$ for every $A_2 \in \mathfrak{A}_2$. If $\mathfrak{A}_i$ is the Borel σ-algebra of some topology $\mathfrak{O}_i$ on $\mathbb{X}_i$, then every continuous function is measurable; the converse does not always hold. The *Cartesian product* of $(\mathbb{X}_1, \mathfrak{A}_1)$ and $(\mathbb{X}_2, \mathfrak{A}_2)$ is the measurable space $(X_1 \times X_2, \mathfrak{A}_1 \otimes \mathfrak{A}_2)$, where $\mathfrak{A}_1 \otimes \mathfrak{A}_2$ is the σ-algebra generated by the sets $A_1 \times A_2$ for $A_i \in \mathfrak{A}_i$. If $\mathbb{X}_1$ and $\mathbb{X}_2$ are disjoint, then the (disjoint) union of the two measurable spaces is the measurable space $(X_1 \cup X_2, \mathfrak{A}_1 \oplus \mathfrak{A}_2)$, where $\mathfrak{A}_1 \oplus \mathfrak{A}_2$ is the set of all sets $A \subseteq X_1 \cup X_2$ such that $A \cap X_i \in \mathfrak{A}_i$. It can be shown that if the spaces $(\mathbb{X}_i, \mathfrak{A}_i)$ are standard Borel spaces then $(X_1 \times X_2, \mathfrak{A}_1 \otimes \mathfrak{A}_2)$ and $(X_1 \cup X_2, \mathfrak{A}_1 \oplus \mathfrak{A}_2)$ are standard Borel spaces as well.

Let $(\mathbb{X}, \mathfrak{A})$ be a measurable space. A *measure* on $(\mathbb{X}, \mathfrak{A})$ is a function M from $\mathfrak{A}$ to $\mathbb{R}_{\geq 0} \cup \{\infty\}$ (the nonnegative reals extended by infinity) that is *σ-additive*, that is, for every countable family $A_1, A_2, \ldots$ of mutually disjoint sets in $\mathfrak{A}$ it holds that $M(\bigcup_{i \geq 1} A_i) = \sum_{i \geq 1} M(A_i)$. A measure M is *finite* if $M(\mathbb{X}) < \infty$, and it is a *probability measure* (or a *probability distribution*) if $M(\mathbb{X}) = 1$. We call $(\mathbb{X}, \mathfrak{A}, M)$ a *measure space*, or a *probability space* if M is a probability measure. $\mathbb{X}$ is called the *sample space* and $\mathfrak{A}$ the *event space* of this probability space.

Example 6. (1) Let $(\mathbb{X}, \mathfrak{A})$ be a measurable space and $Y \subseteq \mathbb{X}$. We define a measure M by letting $M(A)$ be the cardinality of $|A \cap Y|$ (either finite or ∞). M is what we call a *counting measure*. It is finite if Y is finite.
(2) The *normal distribution* $\mathcal{N}(\mu, \sigma)$ is the unique probability measure P on the standard Borel space $(\mathbb{R}, \mathfrak{B})$ with

$$P\big((a, b]\big) = \frac{1}{\sigma\sqrt{2\pi}} \int_a^b \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right) dx.$$

for all $a < b$. (It can be shown that a probability measure on the Borel σ-algebra over the reals is determined by its values on the half open intervals.)
(3) If $\mathbb{X}$ is a countable set (finite or countably infinite), then we can define a probability measure P on $(\mathbb{X}, 2^{\mathbb{X}})$ by defining the singleton values $P(\{x\}) \in [0, 1]$ such that $\sum_{x \in X} P(\{x\}) = 1$ and letting $P(A) \coloneqq \sum_{x \in A} P(\{x\})$ for all $A \subseteq \mathbb{X}$. In fact, every probability measure on $\mathbb{X}$ can be defined this way, regardless of what the σ-algebra is. So for countable probability spaces, we can always assume that the σ-algebra is the power set of the sample space $\mathbb{X}$. ⌟

4. STANDARD PDBS

Let τ be a database schema, and let $\mathbb{U}_\tau$, the *universe*, be the union of the domains of all attributes occurring in τ. We view *database instances* as finite sets or bags (a.k.a. multisets) of *facts* of the form $R(a_1, \ldots, a_k)$, where $R(A_1, \ldots, A_k)$ is a relation schema in τ and for every i the value $a_i \in \mathbb{U}_\tau$ is contained in the domain of attributes A_i. Even if we are only interested in set instances, for technical reasons we need to consider bag instances as well. We denote the set of all facts over τ by $\mathbb{F}_\tau$ and the set of database instances over τ, that is, finite bags of facts in $\mathbb{F}_\tau$, by $\mathbb{DB}_\tau^{\text{bag}}$. Moreover, we denote the subset of all plain sets of facts, that is, set instances, by $\mathbb{DB}_\tau$. In all these notations, we omit the subscript $_\tau$ if the schema is clear from the context or irrelevant.

Under the most general definition, a *probabilistic database* is just a probability space $\mathcal{D} = (\mathbb{D}, \mathfrak{A}, P)$, where $\mathbb{D} \subseteq \mathbb{DB}^{\text{bag}}$. However, it is very difficult to work with this general definition. While we may have an intuition about defining the sample space and the probabilities, it is completely unclear how to define the event space, that is, the σ-algebra $\mathfrak{A}$, which is a set of sets of bags of facts (sic).

Example 7. Recall Examples 1, 3, and let $\mathcal{D} = (\mathbb{D}, \mathfrak{A}, P)$ be the (informally described) PDB shown in Figure 2. The sample space $\mathbb{D} \subseteq \mathbb{DB}_\tau$ consists of all instances

$$\big\{\texttt{Temp(4108,2021-01-05 08:00,}t_1\texttt{)},\ \texttt{Temp(4108,2021-01-05 14:00,}t_2\texttt{)},\ \texttt{Temp(4109,2021-01-05 08:00,}t_3\texttt{)},\ \texttt{Temp(4109,2021-01-05 14:00,}t_4\texttt{)},\ \texttt{Temp(4109,2021-01-06 08:00,}t_5\texttt{)}\big\},$$

where $t_1, \ldots, t_5 \in \mathbb{R}$. We have seen in Example 3 how to calculate the probability of a set of instances. However, it is not obvious which sets are measurable, that is, have a well-defined probability and therefore should belong to the σ-algebra $\mathfrak{A}$. Intuitively, at least sets such as those considered in Example 3 where the temperatures are in certain intervals, should be measurable.

In this simple example, we can define a suitable σ-algebra by an ad-hoc product construction starting from the Borel σ-algebra on the reals, but already in the only slightly more complicated setting of Example 2, where the number of tuples in an instance is also a random variable that is a-priori unbounded, it becomes difficult to carry out such a construction. ⌟

The point is: even if we can somehow come up with an ad-hoc construction of a σ-algebra for every PDB that we want to work with, reasoning about σ-algebras is definitely not what we want to do when working with probabilistic data. Yet, as we have seen in Section 2, measurability is an issue when giving queries and views a meaningful semantics.

A solution to this dilemma is a theoretical framework that gives us a *generic* construction of σ-algebras only depending on the schema τ and the universe $\mathbb{U}$ that is rich enough to make all sets that we typically want to consider measurable and at the same time ensures that all reasonable queries and views are measurable. *Standard probabilistic databases*, introduced in [12], provide such a framework.

The main technical challenge is to generically construct a sufficiently rich σ-algebra $\mathfrak{A} = \mathfrak{A}_\tau$ on $\mathbb{DB}^{\text{bag}}$. If the universe $\mathbb{U}$ is countable, then the set $\mathbb{F}$ of facts and hence the set $\mathbb{DB}^{\text{bag}}$ of all finite bags of facts are countable as well, and we can simply let $\mathfrak{A} = 2^{\mathbb{DB}^{\text{bag}}}$ (see Example 6(3)). But what do we do if the universe is uncountable? The additional assumption we need to make is that we have *a topology on the universe, in fact a Polish topology.* As uncountable universes we may see in typical database applications are usually derived from the reals in some way, this is no serious restriction.

Example 8. Typical domains of database attributes are integers, reals, strings, and time stamps. So we will have a universe like $\mathbb{U} = \Sigma^* \cup \mathbb{R}$ for some finite alphabet Σ (say, UTF8). If by $\mathfrak{O}_\mathbb{R}$ we denote the standard topology on the reals, the generic way of extending it to a Polish topology $\mathfrak{O}$ on $\mathbb{U}$ is to let $\mathfrak{O}$ be the set of all $O \subseteq \mathbb{U}$ such that $O \cap \mathbb{R} \in \mathfrak{O}_\mathbb{R}$. It is straightforward to extend this construction to more complicated universes where we add, for example, a set of (uncountably many) time stamps. ⌟

Let us assume in the following that $\mathfrak{O}_\mathbb{U}$ is a Polish topology on the universe $\mathbb{U}$. We assume that this topology is part of the information provided by the schema τ. Let $\mathfrak{A}_\mathbb{U}$ be the σ-algebra generated by $\mathfrak{O}_\mathbb{U}$. Then $(\mathbb{U}, \mathfrak{A}_\mathbb{U})$ is a standard Borel space. Using finite Cartesian products and disjoint unions, we can lift $\mathfrak{A}_\mathbb{U}$ to a σ-algebra $\mathfrak{A}_\mathbb{F}$ on the set $\mathbb{F}$ of facts. $(\mathbb{F}, \mathfrak{A}_\mathbb{F})$ is still a standard Borel space.

The next step will be to lift $\mathfrak{A}_\mathbb{F}$ to a σ-algebra $\mathfrak{C}$ on $\mathbb{DB}^{\text{bag}}$. Maybe the most direct way of doing this is to first lift the σ-algebra to all finite tuples of facts using finite Cartesian products and a countable disjoint union and then "factor" the resulting σ-algebra through all permutations to go from tuples to bags. A more elegant way of defining the same σ-algebra is as follows.[3] For every set $F \subseteq \mathbb{F}$ of facts and every instance $D \in \mathbb{DB}^{\text{bag}}$, we let $|D|_F$ be the number of elements of F in D counted according to their multiplicities. For example, $|\{\!\{f, f, g, g, g, h\}\!\}|_{\{f,g\}} = 5$. For $n \in \mathbb{N}$, we let $\#(F, n)$ be the set of all $D \in \mathbb{DB}^{\text{bag}}$ with $|D|_F = n$. Finally, we let $\mathfrak{C} \coloneqq \mathfrak{C}_\tau$ be the σ-algebra generated by all sets $\#(F, n)$ for $F \in \mathfrak{C}_\mathbb{F}$ and $n \in \mathbb{N}$. Since $\mathfrak{C}$ is generated by the *counting events* $\#(F, n)$, it is called the *counting σ-algebra* (hence the letter $\mathfrak{C}$ = "Fraktur C"). Another way of seeing $\mathfrak{C}$ is that it is the smallest σ-algebra such that for all measurable sets $F \in \mathfrak{A}_\mathbb{F}$ of facts, the function $|\cdot|_F \colon \mathbb{DB}^{\text{bag}} \to \mathbb{N}$ is measurable with respect to $\mathfrak{C}$ and $2^\mathbb{N}$. We will see that this is enough to guarantee that all queries defined in standard query languages are measurable as well.

[3] It is not obvious that the two constructions indeed lead to the same σ-algebra. This follows from a theorem from point-process theory.

Definition 1. A *standard probabilistic database* is a probability space $(\mathbb{DB}^{\text{bag}}_\tau, \mathfrak{C}_\tau, P)$ for some schema τ.

To keep the definition as simple as possible, we let the sample space of a standard PDB be the set $\mathbb{DB}^{\text{bag}}_\tau$ of all bag instances. As a result, every standard PDB can be specified by its probability distribution. We could adopt a more liberal definition where the sample space consists of an arbitrary measurable subset and then restrict the σ-algebra to this set. That is, we could also admit PDBs of the form $(\mathbb{D}, \mathfrak{C}|_\mathbb{D}, P)$, where $\mathbb{D} \in \mathfrak{C}$ and $\mathfrak{C}|_\mathbb{D} \coloneqq \{C \cap \mathbb{D} \mid C \in \mathfrak{C}\}$. But note that this space is essentially the same as the standard PDB $(\mathbb{DB}^{\text{bag}}, \mathfrak{C}, P')$ where $P'(C) \coloneqq P(C \cap \mathbb{D})$ for all $C \in \mathfrak{C}$. Therefore, it is safe to view such PDBs with restricted sample spaces as standard PDBs. In particular, since the set $\mathbb{DB}$ of all set instances is measurable, this applies to *standard set PDBs* of the form $(\mathbb{DB}, \mathfrak{C}|_{\mathbb{DB}}, P)$.

5. QUERY SEMANTICS

A *view* with *input schema* τ and *output schema* τ' is a mapping $V \colon \mathbb{DB}^{\text{bag}}_\tau \to \mathbb{DB}^{\text{bag}}_{\tau'}$. A *query* is a view where the output schema consists of a single relation. We call a view $V \colon \mathbb{DB}^{\text{bag}}_\tau \to \mathbb{DB}^{\text{bag}}_{\tau'}$ *measurable* if it is a measurable mapping with respect to $\mathfrak{C}_\tau$ and $\mathfrak{C}_{\tau'}$. Such a measurable view V can be lifted to standard PDBs as follows: for every standard PDB $\mathcal{D} = (\mathbb{DB}^{\text{bag}}_\tau, \mathfrak{C}_\tau, P)$, let $V(\mathcal{D})$ be the standard PDB $(\mathbb{DB}^{\text{bag}}_{\tau'}, \mathfrak{C}_{\tau'}, P')$ where P' is defined by

$$P'(C') \coloneqq P\big(V^{-1}(C)\big)$$

for all $C' \in \mathfrak{C}_{\tau'}$. Note that this is exactly semantics defined in (A).

Thus a view has a well-defined semantics on standard PDBs if and only if it is measurable. The following theorem, which is the main result of [12], states that this is the case for a wide class of views.

THEOREM 1. *All queries and views expressible in the relational calculus (with aggregation) and Datalog as well as variants such as Inflationary Datalog and Least Fixed-Point Logic (see* [1]*) are measurable.*

Let us remark that this theorem applies to both set semantics and bag semantics. The proof is a tedious inductive proof that goes through all operators used to define the different query languages. The most involved steps are basic relational-algebra operators such as Cartesian product or projection. The following example exhibits some of the arguments in an easy case that nevertheless already illustrates why we want our underlying topological space to be Polish.

Example 9. Let $\tau = \{R(A, B)\}$, where the attributes A, B have the same domain $\mathbb{U}$. We consider the equality query Q that maps R to its diagonal, that is, the selection

```
SELECT A,B FROM R WHERE A = B.
```

Let d be a metric on $\mathbb{U}$ that induces the Polish topology $\mathfrak{O}_\mathbb{U}$ we assume to exist, and let $Y \subseteq \mathbb{U}$ be a countable dense set. Let $\mathfrak{A}_\mathbb{U}$ be the Borel σ-algebra on $\mathbb{U}$. Then $\mathfrak{A}_\mathbb{U}$ is generated by the open balls $B_{1/n}(y) = \{x \in \mathbb{U} \mid d(x, y) < 1/n\}$ for $y \in Y$ and $n \in \mathbb{N}_{>0}$. Let $\mathfrak{A}_\mathbb{F}$ be the lifted σ-algebra on $\mathbb{F}$,

and $\Delta := \{R(x,y) \in \mathbb{F} \mid x = y\}$ be the diagonal selected by the query Q.

As a first step, we need to prove that $\Delta \in \mathfrak{A}_{\mathbb{F}}$. This is done by characterising its complement $\Delta^{\complement}$. We identify the space $\mathbb{F} = \{R(x,y) \mid x,y \in \mathbb{U}\}$ with the Cartesian product $\mathbb{U} \times \mathbb{U}$ and $\mathfrak{A}_{\mathbb{F}}$ with the product σ-algebra $\mathfrak{A}_{\mathbb{U}} \otimes \mathfrak{A}_{\mathbb{U}}$ generated by the sets $A \times A'$ for $A, A' \in \mathfrak{A}_{\mathbb{U}}$. Then, Δ becomes $\{(x,x) \mid x \in \mathbb{U}\}$. Observe that for $(x,x') \in \Delta^{\complement}$, there are $y, y' \in Y$ and an $n \in \mathbb{N}_{>0}$ such that $(x,x') \in B_{1/n}(y) \times B_{1/n}(y')$ and $\big(B_{1/n}(y) \times B_{1/n}(y')\big) \cap \Delta = \emptyset$. Thus

$$\Delta^{\complement} = \bigcup_{\substack{y,y' \in Y \\ B_{1/n}(y) \times B_{1/n}(y') \cap \Delta = \emptyset}} B_{1/n}(y) \times B_{1/n}(y'),$$

which is a countable union of sets in $\mathfrak{A}_{\mathbb{U}} \otimes \mathfrak{A}_{\mathbb{U}}$. Since every σ-algebra is closed under complementation and countable intersections, it follows that $\Delta \in \mathfrak{A}_{\mathbb{U}} \otimes \mathfrak{A}_{\mathbb{U}}$.

To prove that the query Q, formally a mapping from $\mathbb{DB}_{\tau}^{\text{bag}}$ to $\mathbb{DB}_{\tau}^{\text{bag}}$, is measurable, we need to prove that for every $C \in \mathfrak{C}$ the pre-image $Q^{-1}(C)$ is in $\mathfrak{C}$ as well. As the counting events $\#(F,n)$ for $n \in \mathbb{N}$ and $F \in \mathfrak{A}_{\mathbb{F}}$ generate $\mathfrak{C}$, it suffices to prove that the pre-image of each such counting event is in $\mathfrak{C}$. Observe that for every instance $D \in \mathbb{DB}^{\text{bag}}$ we have $Q(D) \in \#(F,n)$ if and only if D contains exactly n facts $R(x,y) \in F$ with $x = y$ (counted according to multiplicity), or equivalently, $R(x,y) \in F \cap \Delta$. That is,

$$Q^{-1}(\#(F,n)) = \#(F \cap \Delta, n).$$

$F, \Delta \in \mathfrak{A}_{\mathbb{F}}$ imply $F \cap \Delta \in \mathfrak{A}_{\mathbb{F}}$ and thus $\#(F \cap \Delta, n) \in \mathfrak{C}$. ⌟

6. REPRESENTATIONS

An infinite PDB viewed as a probability distribution over database instances is an idealised mathematical concept that allows us to give semantics to PDBs and queries. It is not something that we can ever materialise. When designing probabilistic database systems, we need to think about finite representations of the probability spaces.

The most common model for finite probabilistic databases is that of *tuple-independent (TI)* PDBs. We can adopt the notion of TI PDBs to countable PDBs; to represent a countably infinite TI PDB we only need to represent a function that assigns a probability to every fact. Countably infinite TI PDBs were studied in [10]. An extension to uncountable PDBs, called *Poisson PDBs*, was proposed in [11]. Another basic model, *block-independent disjoint (BID)* PDBs, can also be extended to the infinite setting [10, 11]. Both TI and BID PDBs can only represent very simple probability distributions. To obtain more sophisticated distributions, we can apply transformations such as views to such PDBs (views of countable TI and BID PDBs were studied in [6]) or combine several PDBs into a new one using constructions such as convex combinations and superpositions (see [11]).

A different and more general approach is to start from a deterministic set of data, feed it into some generative model, and interpret the output as a probability distribution on database instances. With this approach, we are free to use all kinds of probabilistic modelling formalisms, for example, database-valued Markov processes [13], logical formalisms such as Markov Logic Networks [18] or ProbLog [8], deep neural network models such as variational autoencoders [15], or programs in some probabilistic programming language [4]. The difficulty is to specify such models in a way that the output can be interpreted as a meaningful probability distribution on database instances. Generative Datalog (GDatalog), introduced in [3, 9], is a declarative probabilistic programming language that remains within the framework of relational databases and therefore avoids this difficulty; the output of a Generative Datalog program is a PDB by definition.

7. GENERATIVE DATALOG

Throughout this section, we assume a set semantics for relational databases. For recursive languages like Datalog, a bag semantics is less convenient, because we want to avoid repeatedly generating new copies of the same fact.

We start by informally reviewing Datalog (see [1] for more background). A *Datalog program* is a finite set of *rules* of the form

$$R(\overline{x}) \leftarrow S_1(\overline{x}_1), \ldots, S_m(\overline{x}_m), \tag{B}$$

where R, S_i are relation symbols and $\overline{x}, \overline{x}_i$ are tuples of variables of the appropriate lengths such that all variables in the tuple $\overline{x}$ appear in one of the tuples $\overline{x}_i$. The *head* of the rule (B) is $R(\overline{x})$ and the *body* is $S_1(\overline{x}_1), \ldots, S_m(\overline{x}_m)$. The relations appearing in the head of some rule of a Datalog program are *intensional*; all other relations are *extensional*. The extensional (intensional) relations form the *extensional* (*intensional*, resp.) *schema* of the program.

Consider the rule (B). Given an interpretation of all the body relations S_i and an assignment α to the variables $\overline{x}_i$, the rule is applicable if for all i the fact $S_i(\alpha(\overline{x}_i))$ holds true under the current interpretation of S_i. The application of the rule generates the new fact $R(\alpha(\overline{x}))$.

We run a Datalog program $\mathcal{P}$ on a database instance over the extensional schema. The program iteratively computes interpretations for all the intensional relation symbols. All intensional relations are initialised to be empty. Then the rules of the program are applied repeatedly until no more new facts can be generated. It can be shown that the final relations do not depend on the order in which the rules are applied. Furthermore, the program (when applied to a finite input instance) always terminates in finitely many steps and hence the result can be interpreted as a database instance over the intensional schema. In other words, a Datalog program expresses a view mapping instances over the extensional schema to instances over the intensional schema.

The following example illustrates these Datalog definitions and then develops the main ideas of its probabilistic extension *GDatalog*.

Example 10. Let $\mathcal{P}$ be the following simple Datalog program with extensional relations S, E and an intensional relation R:

$$R(x) \leftarrow S(x), \tag{C}$$
$$R(x) \leftarrow R(y), E(y,x). \tag{D}$$

We can interpret instances over the extensional schema as directed graphs with a distinguished set of source vertices. Then the program computes the set of all vertices reachable from the source vertices.

Now suppose we do not only want to compute whether a vertex is reachable from a source, but also how long it takes to reach it. Assume that the edge relation is now ternary, where we interpret the third, real valued component as a

length or traversal time. Consider the following program:

$$R(x,0) \leftarrow S(x), \tag{E}$$

$$R(x,t+s) \leftarrow R(y,t), E(y,x,s). \tag{F}$$

This is no longer a Datalog program in the strict sense. Yet it seems clear what its semantics is; after executing the program, the binary relation R will contain all pairs (x,t) such that x is reachable from a source vertex by a walk (that is, a path with possibly repeated vertices and edges) of length t. There is however a problem with this: if the graph is cyclic, there may be arbitrarily long walks, and the output relation will no longer be finite. Therefore, let us assume that the input graph is acyclic.

Now assume the traversal time of an edge is not deterministic, but random. Say, we model it by a log-normal distribution $\mathcal{LN}(\ln s, 1/10)$ where the parameter s is the median of this distribution. We may write the following program:

$$R(x,0) \leftarrow S(x), \tag{G}$$

$$R(x,t+\mathcal{LN}(\ln s, 1/10)) \leftarrow R(y,t), E(y,x,s). \tag{H}$$

The output of this program is supposed to be a random relation that contains pairs (x,t), where t is a sampled travel time along some walk from a source to x in the input graph. We can interpret the probability space of all possible output relations as a PDB of schema $\{R\}$. Our *GDatalog program* applied to an acyclic input graph thus represents a PDB.

But now another problem with termination pops up, even if the input graph is acyclic. The intuitive interpretation of an application of rule (H) is that for x,t,s,y matching the body of the rule, we sample a value r from the log-normal distribution $\mathcal{LN}(\ln s, 1/10)$ and then generate the fact $R(x,t+r)$. But if we would apply the same rule again to the same x,t,s, almost surely we would not sample the same r again, but an $r' \neq r$, and hence generate a new fact $R(x,t+r')$. We could do this over and over again and would obtain an infinite relation R, and moreover, the program would never terminate. This is clearly not what we want. Note that this cannot happen with a deterministic rule like (F).

Our simple solution to avoid this problem is to stipulate that *we can only apply the rule once to every triple (x,t,s) of parameters*. There may, however, be application scenarios where it is desirable to sample from the distribution more than once. To accommodate this, *we allow the same rule to appear several times in a program*. Then for each instantiation of the rule, we can sample once. A more flexible, but more complicated way of sampling several times with the same parameter tuple is to introduce another parameter that serves as an index for the samples. ⌟

Rules (G) and (H) give a typical example of a GDatalog program. To define GDatalog programs in general, besides the extensional and intensional schema we need to specify a family Ψ of *parametrised distributions*. An example is the log-normal distribution $\mathcal{LN}(\mu,\sigma)$ with parameters the real number μ and the positive real number σ. It may be helpful to think of parameterised distributions as randomised functions mapping the parameters, such as μ and σ, to values in some range, in the case of $\mathcal{LN}(\mu,\sigma)$ the positive reals. As a second example, consider the simple Bernoulli (coin-flip) distribution $\mathcal{B}(p)$ with parameter $p \in (0,1)$. It takes value 1 with probability p and 0 with probability $1-p$. The functions in Ψ must satisfy some technical measurability conditions to ensure that they behave well with respect to changes of parameters. Intuitively, we want continuous changes in the parameters to result in continuous changes of the distribution, whatever that means technically. As examples, think of a normal distribution $\mathcal{N}(\mu,\sigma)$ and a Bernoulli distribution $\mathcal{B}(p)$. We can compose the parameterised distributions and replace parameters by constants to form more complex terms, but we need to make sure that the resulting parameterised distributions still satisfy our technical conditions. We call these *Ψ-terms*. An example of such a term, with two variables s,t, is the expression $t+\mathcal{LN}(\ln s, 1/10)$ in rule (H). Deterministic functions such as $t+s$ can be easily incorporated into the semantics as well. A *GDatalog rule* over a set Ψ of parametrised distributions is an expression

$$R(\bar{t}) \leftarrow S_1(\bar{x}_1), \ldots, S_m(\bar{x}_m), \tag{I}$$

where the *body* $S_1(\bar{x}_1), \ldots, S_m(\bar{x}_m)$ is a list of atoms over the extensional and intensional schema, just like for normal Datalog rules, and the *head* $R(\bar{t})$ consists of an intensional relation symbol R and a tuple $\bar{t} = (t_1, \ldots, t_k)$ of (Ψ-)terms such that all variables of the t_i appear in the body of the rule. Of course we must make sure that the terms are of the appropriate types, that is, the range of t_i is contained in the domain of the ith attribute of relation R. Note that in particular, all normal Datalog rules are GDatalog rules.

A *GDatalog program* is a bag of GDatalog rules.

Before even touching upon the intricacies of a formal semantics for GDatalog programs, let us explain an informal operational semantics for GDatalog rules and programs. We have already given the intuition in Example 10. Consider the rule (I). Let $\bar{t} = (t_1, \ldots, t_k)$, and let $\bar{y}_i$ be the tuple of variables of the term t_i—we indicate this by writing $t_i(\bar{y}_i)$. Note that for Ψ-terms, $t_i(\bar{y}_i)$ is a parametrised probability distribution: if we instantiate the variables in $\bar{y}_i$ by values of the appropriate type, we obtain a probability distribution.

Given an interpretation of all the body relations S_i and an assignment α to the variables $\bar{x}_i$, the rule is applicable if for all i the fact $S_i(\alpha(\bar{x}_i))$ holds true under the current interpretation of S_i. To apply the rule, for all j we sample a value a_j from the probability distributions $t_j(\alpha(\bar{y}_j))$ to generate the new fact $R(a_1, \ldots, a_k)$.

We run a GDatalog program $\mathcal{G}$ on a database instance over the extensional schema in a similar way as a normal Datalog program. All intensional relations are initialised to be empty. By repeatedly applying the rules as described above, the program generates (random) facts. All rule applications are stochastically independent. We stipulate that each rule of the program (or more precisely, each occurrence of each rule—remember a program is a bag of rules where a rule may occur several times) can only be applied once for every instantiation of the variables appearing in the head of the rule. The computation terminates if no more rule can be applied, and the output is the set of facts generated by the program during the computation, that is, a database instance over the intensional schema. Because of the sampling of values in the rule applications, the output is probabilistic. We interpret it as a probabilistic database. Thus, given a database instance over the extensional schema, a GDatalog program generates a PDB over the intensional schema.

However, our informal description of the semantics raises several crucial questions:

(1) Does the program always terminate?
(2) How can we be sure that the output is indeed a well-

defined probabilistic database?

(3) In which order do we apply the rules, and does this make a difference?

The answer to Question (1) is simply 'no' (in general). In Example 10, we already saw that even the simple deterministic program (E), (F) may not terminate if the input graph is cyclic. For probabilistic programs, the notion of termination is more complicated, because a program may terminate for certain random outcomes while it diverges for other outcomes [5]. We resolve this issue by conditioning the output probability distribution on termination. That is to say, a GDatalog program defines a *sub-probabilistic database* where the probability mass over its defined space may be smaller than 1. It is an open research question to understand termination criteria for GDatalog programs.

We can answer Questions (2) and (3) by carefully defining a formal semantics for GDatalog programs. The main results of [9] (in the case of discrete probability distributions due to [3]) regarding this semantics are (informally) summarised in the following theorem.

THEOREM 2. *Applying the GDatalog program $\mathcal{G}$ to a database instance D over the extensional schema, defines a standard sub-probabilistic database $\mathcal{G}(D)$.*

$\mathcal{G}(D)$ does not depend on the order in which the rules are applied, as long as the policy that is used to decide which rules to apply is measurable.

One final remark is that the semantics of GDatalog programs can be lifted to probabilistic databases. That is, if we apply a GDatalog program to a standard PDB over the extensional schema it defines a standard sub-probabilistic database over the extensional schema.

8. CONCLUDING REMARKS

To enable reasoning about uncertain data with standard statistical models, we need probabilistic databases to support continuous probability distributions. Providing a mathematical framework for dealing with a very general class of probability distributions, we introduced standard PDBs [12]. Furthermore, we extended Generative Datalog [3], a declarative probabilistic programming language for relational data, to continuous distributions, thereby providing a flexible formalism for specifying generative models of standard probabilistic databases.

The focus of our work was on semantical issues. Further research is needed to address algorithmic and complexity theoretic questions.

Acknowledgements. This research is supported by the German Research Foundation (DFG) under grant GR 1492/16-1 and the Research Training Group 2236 UnRAVeL.

9. REFERENCES

[1] S. Abiteboul, R. Hull, and R. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] P. Agrawal and J. Widom. Continuous Uncertainty in Trio. In *Proc. VLDB Workshop on Management of Uncertain Data*, pages 17–32, 2009.

[3] V. Bárány, B. ten Cate, B. Kimelfeld, D. Olteanu, and Z. Vagena. Declarative Probabilistic Programming with Datalog. *ACM Transactions on Database Systems (TODS)*, 42(4), 2017.

[4] G. Barthe, J.-P. Katoen, and A. Silva, editors. *Foundations of Probabilistic Programming*. Cambridge University Press, 2020.

[5] O. Bournez and F. Garnier. Proving positive almost-sure termination. In *RTA*, volume 3467 of *Lecture Notes in Computer Science*, pages 323–337. Springer, 2005.

[6] N. Carmeli, M. Grohe, P. Lindner, and C. Standke. Tuple-independent representations of infinite probabilistic databases. *ArXiv*, 2008.09511, 2020.

[7] D. J. Daley and D. Vere-Jones. *An Introduction to the Theory of Point Processes, Volume I: Elementary Theory and Models*. Probability and Its Applications. Springer, 2nd edition, 2003.

[8] L. De Raedt, A. Kimmig, and H. Toivonen. ProbLog: A Probabilistic Prolog and Its Application in Link Discovery. In *Proc. IJCAI 2007*, pages 2468–2473.

[9] M. Grohe, B. L. Kaminski, J.-P. Katoen, and P. Lindner. Generative datalog with continuous distributions. In *Proc. PODS 2020*, pages 347–360, New York, NY, USA.

[10] M. Grohe and P. Lindner. Probabilistic Databases with an Infinite Open-World Assumption. In *Proc. PODS 2019*, pages 17–31.

[11] M. Grohe and P. Lindner. Independence in infinite probabilistic databases. *ArXiv*, 2011.00096, 2020.

[12] M. Grohe and P. Lindner. Infinite Probabilistic Databases. In *Proc. ICDT 2020*, pages 16:1–16:20, 2020.

[13] R. Jampani, F. Xu, M. Wu, L. Perez, C. Jermaine, and P. J. Haas. The Monte Carlo Database System: Stochastic Analysis Close to the Data. *ACM Transactions on Database Systems (TODS)*, 36(3):18:1–18:41, 2011.

[14] O. Kennedy and C. Koch. PIP: A Database System for Great and Small Expectations. In *Proc. ICDE 2010*, pages 157–168.

[15] D. P. Kingma and M. Welling. An introduction to variational autoencoders. *Foundations and Trends in Machine Learning*, 12(4):307–392, 2019.

[16] O. Macchi. The Coincidence Approach to Stochastic Point Processes. *Advances in Applied Probability*, 7(1):83–122, 1975.

[17] J. E. Moyal. The General Theory of Stochastic Population Processes. *Acta Mathematica*, 108:1–31, 1962.

[18] M. Richardson and P. Domingos. Markov logic networks. *Machine Learning*, 62(1–2):107–136, 2006.

[19] S. Singh, C. Mayfield, R. Shah, S. Prabhakar, S. Hambrusch, J. Neville, and R. Cheng. Database Support for Probabilistic Attributes and Tuples. In *Proc. ICDE 2008*, pages 1053–1061.

[20] D. Suciu. Probabilistic databases for all. In *Proc. PODS 2020*, pages 19–31, 2020.

[21] D. Suciu, D. Olteanu, C. Ré, and C. Koch. *Probabilistic Databases*. Synthesis Lectures on Data Management. Morgan & Claypool, 2011.

[22] G. Van den Broeck and D. Suciu. Query Processing on Probabilistic Data: A Survey. *Foundations and Trends® in Databases*, 7(3–4):197–341, 2017.

Technical Perspective for: Query Games in Databases

Dan Suciu
University of Washington

When a data analyst runs some query to analyze her data, she often wants to ask some follow-up questions, about the result of the query. *Why*-questions take many shapes, and occur in many scenarios. Why is a particular tuple in the answer? Why is it *not* in the answer? Why is this graph decreasing? Why did we observe a sudden burst of error messages in online monitoring? Database researchers have noted the need for *why*-questions, and the literature contains several approaches, mostly tailored to specific applications. Despite the interest and the work in this area, there is currently no consensus of what an explanation to a query answer should be, and how one should compute it.

There are two challenges in addressing *why*-questions. The first is to decide the type of the explanation, in other words what to return in response to *why?* The system may return a particular tuple, or a predicate defining a subset of the data, or just an attribute name. This challenge is best addressed with techniques from psychology and HCI, since the choice of the explanation type is ultimately evaluated by end users. The second challenge is to define a quantitative degree of explanation: given the list of all potential explanations, compute a quantitative score, and present them to the user ranked by this score. This challenge requires both a formal definition of the score, and the design of an algorithm to compute that score.

The paper *Query Games in Databases* by Livshits et al. proposes an elegant numerical definition of an explanation, based on a well known concept in economics, called the Shapley value of cooperative games. While originally proposed in economics, the Shapley value has been applied to a variety of domains, such as game theory, political science, and risk analysis. I would also encourage the reader to check out Shapley's original paper from 1952, which is short and remarkably accessible, and available in the survey [33]. Next, Livshits' paper adopts the Shapley value for explaining query answers: the players are the tuples in the database, and the "game" is the query answer. This definition associates a numerical score to every tuple in the database, which represents the tuple's contribution to the observed query output. The paper illustrates with several examples, then focuses on algorithmic aspects of computing this score. As with other definitions of explanation, computing the Shapley value is harder than computing the query answer itself. In fact, the authors prove that it is hard for #P in general, and characterize the queries for which this score can be computed in polynomial time or is #P-hard. They also describe an approximation algorithm, which is of practical interest.

One major attraction of the Shapley value is that it is uniquely defined by three simple axioms. In other words, the Shapley value is the unique scoring function that satisfies three, very natural properties, which Shapley called *symmetry*, *efficiency*, and *aggregation*. Thus, a definition of explanation based on the Shapley value feels principled, in contrast to other definitions of explanation that were mostly inspired by influence analysis or by the responsibility score in causal analysis, and feel more ad-hoc.

Stepping back to look at the bigger picture, the problem studied in this paper is related to *explainable AI*, whose goal is to make automated decisions based on machine learning models transparent to the end users. Researchers have adapted the Shapley value to that setting too: the units of explanation are the features of the ML model, and the goal of the explanation is to associate a score to each feature, quantifying its contribution to the output of the classifier and, like in Livshits' approach, computing the explanation score turns out to be significantly harder than computing the classifier's output. When viewed from this perspective, it becomes clear that Livshits' paper makes a foundational contribution to the general quest of explaining automated decisions.

Query Games in Databases

Ester Livshits
Technion, Haifa, Israel
esterliv@cs.technion.ac.il

Leopoldo Bertossi
Univ. Adolfo Ibáñez and
Millennium Inst. Foundations
of Data (IMFD), Chile
leopoldo.bertossi@uai.cl

Benny Kimelfeld
Technion, Haifa, Israel
bennyk@cs.technion.ac.il

Moshe Sebag
Technion, Haifa, Israel
moshesebag@campus.technion.ac.il

ABSTRACT

Database tuples can be seen as players in the game of jointly realizing the answer to a query. Some tuples may contribute more than others to the outcome, which can be a binary value in the case of a Boolean query, a number for a numerical aggregate query, and so on. To quantify the contributions of tuples, we use the Shapley value that was introduced in cooperative game theory and has found applications in a plethora of domains. Specifically, the Shapley value of an individual tuple quantifies its contribution to the query. We investigate the applicability of the Shapley value in this setting, as well as the computational aspects of its calculation in terms of complexity, algorithms, and approximation.

1. INTRODUCTION

In data management, so as in artificial intelligence (AI), there is an increasing need for characterizing and computing explanations of the outcomes of algorithms. In AI, this is commonly attempted for classification algorithms. In relational databases, query answering may be the operation that requires explanations. As we will see later on, the two areas share concerns and approaches. On the databases side, we may want to explain *why* we obtained a particular answer to a query; or why we *did not* get some other answer we may have in mind. As expected, the explanations depend on the underlying data and the query itself. (We assume that the query-answering algorithm is correct.) In this work, we concentrate on the positive case, that is, on why an answer was obtained (as opposed to not obtained).

Explanations can be given in terms of individual tuples of the underlying database that contribute to the answer or in terms of the *provenance* or *lineage* of the query [10, 18, 35], that explicitly describe the logical connection between a generic query answer and the possible tuples of the database. In this sense, provenance implicitly describes *how* a query answer can be obtained. In this work, we focus mostly on the identification of individual tuples that contribute to a query answer, through, and accompanied by, their quantitative "degree of contribution."

 This paper is based on the paper entitled "The Shapley Value of Tuples in Query Answering", published in 23rd International Conference on Database Theory (ICDT 2020). Editors: Carsten Lutz and Jean Christoph Jung; Article No. 20; pp. 20:1–20:19. Leibniz International Proceedings in Informatics Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany. DOI: https://doi.org/10.4230/LIPIcs.ICDT.2020.20

Example 1. Consider the following relational database D, with the table *Store* representing official stores, and the table *Receives* for stores receiving goods from other stores:

Receives	receiver	sender
	s_2	s_1
	s_3	s_3
	s_4	s_3

Store	store
	s_2
	s_3
	s_4

For accounting purposes, a store could be its own supplier, as shown by the tuple $Receives(s_3, s_3)$. The query that asks whether there are pairs of official stores in a receiving relationship is expressed (in predicate logic) as

$$\mathcal{Q}\colon\ \exists x \exists y (Store(x) \wedge Receives(x, y) \wedge Store(y)) \qquad (1)$$

and we use $D \models \mathcal{Q}$ to denote that it holds true in the database D. This is a *conjunctive query* (CQ), as it is built as a conjunction of *atoms* (database predicates instantiated on constants or variables) preceded by a sequence of existential quantifiers.

The question is about the tuples (i.e., rows in tables) that cause this query to be true (or lead to the answer *yes*). In this case, among the tuples that contribute to the answer we find the tuples $Store(s_4)$, $Receives(s_4, s_3)$ and $Store(s_3)$ that, taken together, make $\mathcal{Q}$ evaluate to true. □

In Example 1 we informally used the notion of a *cause*. Actually, there is a precise notion of a tuple as an *actual cause* for a query answer [26, 27]. This notion was borrowed from a more general notion of actual causality [20] that defines causes in terms of counterfactual interventions (cf., e.g., [19]). These are hypothetical, exploratory updates on a variable of a (structural) model that are made for detecting whether the output changes. For monotone queries, such as (1) whose answer set can only grow when the database grows, the relevant interventions are tuple deletions and changes of attribute values. In this work, we stick to the former. Conjunctive queries are always monotone, and so are *unions (disjunctions) of conjunctive queries* (UCQs).

Actual causality can be extended by means of a measure of *causal responsibility* that quantifies the strength of an actual cause for the outcome at hand [12]. Responsibility can be applied to database tuples, to capture in quantitative terms the relevance of a tuple for the query result [9, 27]. We introduce and illustrate actual causality and responsibility on our running example.

Example 2. (ex. 1 cont.) The tuple $Store(s_3)$ is a *counterfactual cause* for $\mathcal{Q}$ (to be true in D), because: (a) $D \models \mathcal{Q}$, and (b) $D \smallsetminus \{Store(s_3)\} \not\models \mathcal{Q}$; that is, $\mathcal{Q}$ is no longer true once $Store(s_3)$ is removed.

Now, $Receives(s_4, s_3)$ is an *actual cause*, because: (a) $D \models \mathcal{Q}$, and (b) there is $\Gamma \subseteq D$ with $Receives(s_4, s_3) \notin \Gamma$, such that $D \smallsetminus \Gamma \models \mathcal{Q}$, but $D \smallsetminus (\Gamma \cup \{Receives(s_4, s_3)\}) \not\models \mathcal{Q}$. In this case, Γ is a *contingency set* for $Receives(s_4, s_3)$, and $\Gamma = \{Receives(s_3, s_3)\}$ is a minimum-size contingency set. It holds $D \smallsetminus \{Receives(s_4, s_3), Receives(s_3, s_3)\} \not\models \mathcal{Q}$.

We can see that the counterfactual cause $Store(s_3)$ is also an actual cause, with the empty set as the minimum-size contingency set: it does not need any company to invalidate the query when removed from the database.

Causal responsibility is defined in terms of minimum-size contingency sets: If a tuple τ is an actual cause for $\mathcal{Q}$, its responsibility is $\rho(\tau) = \frac{1}{1+|\Gamma|}$, where Γ is a minimum-size contingency set for τ. The responsibility is set to 0 when a tuple is not an actual cause.

We can see then that $Store(s_3)$ is an actual cause with responsibility 1; and the tuple $Receives(s_4, s_3)$ is an actual cause with responsibility $\frac{1}{2}$. Similarly, $Receives(s_3, s_3)$ and $Store(s_4)$ are actual causes, each with responsibility $\frac{1}{2}$. □

As earlier said, the responsibility score can be assigned to variables, or values of variables, that participate in a model that contains output variables. In databases, tuples can be seen as binary variables taking values 1 or 0, indicating whether or not the tuple belongs to the database, respectively. At the same time, these variables are the input of a model that captures both the database and the query at hand. A probability-based generalization of the responsibility score has also been applied to assign (numerical) scores to values of features that characterize entities that are subject to classification by means of a trained classification model [7]. This can be done with either black-box or open models. This way, it is possible to identify the feature values that are most relevant to the outcome of the classification.

As we have seen, the contribution of tuples to query results has been based on causality-based approaches, and indirectly, on the concept of explanation. Actually, explanations have been treated in many disciplines, and, in particular, in AI, under *model-based diagnosis* [34], where *consistency-based* and *abductive* are the main approaches. Some connections between actual causality in databases and both forms of diagnosis have been established in [8, 9], respectively.

Back to databases, Salimi et al. [31] have illustrated that in some situations, responsibility may assign non-intuitive scores to tuples. As an alternative, they introduced the *causal-effect* score. This goes through creating a probabilistic database [35], seeing the query as a binary random variable, and computing the difference between the expected values of the query conditioned on the presence and the absence of the tuple under consideration. Their examples showed that the causal-effect score better captures the intuition in some cases. Yet, their effort brings up some basic questions. Is there any notion that directly considers and quantifies the intuitive notion of *contribution*? What makes the choice of a contribution score a good one? Fortunately, questions of this sort have been addressed in decades of research in the field of game theory.

Indeed, in this work, we treat the contribution from the viewpoint of game theory and tie it to the question of *how to properly distribute wealth (profit) among collaborating agents*. To this end, we appeal to widely applied and established concepts and techniques from *cooperative game theory*. There, we find as a natural candidate the popular *Shapley value*, introduced by Lloyd Shapley in 1953 [32] as a measure of the contribution of a player to the common wealth associated with a multiplayer coalition game.

In our setting, the database tuples can be seen as the players, and the query value as the numerical, joint wealth function. The query can be Boolean, taking the values 1 or 0, for true and false, respectively. This is the case for the query in (1). It could also be an aggregate numerical query that maps the database into a number.

Example 3. (ex. 1 cont.) We add a third attribute (column), amount, to the table *Receives*, indicating the amount of money received by the store in the first column from that in the second. We also add the last two tuples, obtaining a new database instance D'.

Receives'	receiver	sender	amount
	s_2	s_1	10
	s_3	s_3	25
	s_4	s_3	15
	s_2	s_4	18
	s_2	s_3	20

We can pose the query about the total amount received by store s_2 from official stores:

$$\mathcal{Q}_1\colon\ sum\{\!\{y \mid \exists z(Receives'(s_2, z, y) \wedge Store(z))\}\!\}. \qquad (2)$$

Here, we use the bag notation $\{\!\{\cdot\}\!\}$ since we care about all of the numbers y and not just the distinct ones. Only the last two tuples of *Receives'* contribute to the sum (not the first one, because s_1 is not an official store). So, the query answer is $18 + 20 = 38$. Similarly, we can pose the query about the maximum amount received by s_2:

$$\mathcal{Q}_2\colon\ max\{z \mid \exists y(Receives'(s_2, y, z) \wedge Store(y))\}, \qquad (3)$$

with answer 20. Notice that these are aggregations over a conjunctive query.

In this work, we consider only *scalar aggregate queries*, i.e. without *group-by*. □

Using the Shapley value, we can quantify the contribution of tuples to query answers. For example, the contribution of the tuple $Store(s_3)$ to the answer, 1, of the query (1), will be $\mathsf{Shapley}_{\mathcal{Q}}(D, Store(s_3))$, that is, the Shapley value of the tuple $Store(s_3)$ of database D w.r.t. the query $\mathcal{Q}$. Similarly, we can quantify the contribution of the tuple $Receives'(s_2, s_4, 18)$ of D' to the query $\mathcal{Q}_1$, through $\mathsf{Shapley}_{\mathcal{Q}_1}(D', Receives'(s_2, s_4, 18))$.

As we will see in Section 2, the general definition of the Shapley value has a counterfactual flavor, in that interventions on the players are implicitly considered. As in actual causality in databases [27], our application of the Shapley value allows to partition the database into *endogenous* and *exogenous* tuples. Only for the former, we quantify the degree of contribution to query answering, whereas the latter are taken as given and fixed. They are beyond our control or scope of analysis. They could be, for example, tuples inherited from external sources or legacy data. The partition is application dependent. For the same reason, exogenous

tuples are not subject to counterfactual interventions, in particular, to hypothetical deletions, and are always present.

Given the set of players and the "wealth function", the definition of the Shapley value of a player follows a general pattern (see Section 2). It is also well known that the Shapley value possesses properties that cast it as natural and intuitive. Actually, the Shapley value emerged as the only function that enjoys those desirable properties [33].

If we decide to adopt this approach, the main general question in our context is: *What is the computational cost of computing the Shapley value of a tuple, for a fixed, given query?* Since the query is fixed, this question is about *data complexity*, that is, the time complexity of computing the Shapley value for a tuple in terms of the size of the underlying database D. We should mention that, although we are computing the Shapley value for a single, particular tuple, all the other players (tuples) have to be taken into account.

Interestingly, the answer to the above question depends on the syntax of the query. We can provide a fairly complete picture of the complexity of the Shapley value computation when the query is conjunctive, or an aggregation over a conjunctive query. However, with a proviso: as long as the query does not contain self-joins. For example, query (1) contains a self-join since the predicate *Store* appears twice in the conjunction. It is important to notice that CQs with self-joins have turned out to be elusive when it comes to fully characterizing the complexity of dealing with them in several different data management tasks.

For CQs without self-joins, we can give a precise characterization of the complexity of computing the Shapley value. Even more, we can provide a *dichotomy result* that tells us that CQs of a certain syntactic form can be computed in polynomial time (in the size of the database D), and any other CQ is provably *hard* to compute, in a precise sense, as we will see in Section 4. This classification can be extended to some aggregations over self-join free CQs.

Example 4. (ex. 3 cont.) The purely conjunctive part of query (2), namely

$$\mathcal{Q}_1'\colon\ \exists y \exists z (\mathit{Receives}'(s_2, z, y) \wedge \mathit{Store}(z)). \tag{4}$$

is self-join free. Just by looking at its syntactic shape, we can conclude that the Shapley value of any of the tuples in the database D' can be computed in polynomial time in the size of D'.

Now, let us add to the database a new table, *IntStore*, displaying international stores. We denote the resulting database by D''.

IntStore	istore
	s_2
	s_3
	s_5

Now, the query is about international stores in receiving relationship with official stores:

$$\mathcal{Q}_3\colon\ \exists x \exists y \exists z (\mathit{IntStore}(x) \wedge \mathit{Receives}'(x, y, z) \wedge \mathit{Store}(y)). \tag{5}$$

This query is self-join free. Again, from the syntactic shape of this formula, we can conclude that computing the Shapley values of database tuples is computationally hard. □

Actually, the main result in this work is a dichotomy criterion that tells us that, for a self-join free CQ $\mathcal{Q}$, the following holds: If $\mathcal{Q}$ is *hierarchical* (as we formally define in Section 4), then $\mathsf{Shapley}_{\mathcal{Q}}(D, \tau)$, for $\tau \in D$, can be computed in polynomial time in the size, $|D|$, of D. Otherwise, the computation is #P-complete, which can be interpreted as an intractable complexity class. The hierarchy condition is purely syntactic and can be easily checked. This result can be extended to some of the aforementioned aggregations.

It is worth mentioning here that this dichotomy result follows the same pattern as that for answering conjunctive queries over *probabilistic databases* [15, 35]. In particular, the same hierarchy condition is used. However, the proofs for the probabilistic case cannot be used or easily adapted to our case. We take a fresh route, at least for the harder part of the proof (namely, the hardness part of the dichotomy).

Another interesting result we obtain states the existence of an efficient and good approximation algorithm for the hard cases of the computation of the Shapley value. In particular, every UCQ (and some aggregations over UCQs) has efficient approximation algorithms for arbitrary approximation ratios.

In this section, we discussed mainly Boolean CQs and unions thereof, where the answer to the query is 1 or 0, and scalar, numerical aggregations over CQs. In this way, the queries always return a number. The extension of the results presented here to *open queries* with variables, such as $\mathcal{Q}(z)\colon\ \exists y(\mathit{Receives}'(s_2, z, y) \wedge \mathit{Store}(z))$, asking now about official stores, is rather straightforward: one keeps fixed a particular answer, instantiates the query with it, obtaining a Boolean query, and then one proceeds as before. (More details are given in Section 4.)

Many more results than those presented here, and their technical details, can be found in the conference version of this article [23].

2. THE SHAPLEY VALUE OF TUPLES

Let us consider a set of players D, and a *wealth function* (or simply, *game function*) $\mathcal{G}$ that assigns real numbers to the subsets of the players, that is, $\mathcal{G} : \mathcal{P}(D) \longrightarrow \mathbb{R}$. Here, $\mathcal{P}(D)$ denotes the power set of D (i.e., the set of all subsets of D). The contribution of a particular player $p \in D$ to the common wealth of the game represented by $\mathcal{G}$ is its Shapley value, defined by:

$$\mathsf{Shapley}_{\mathcal{G}}(D, p) \stackrel{\text{def}}{=\!=} \sum_{S \subseteq D \setminus \{p\}} \frac{|S|!(|D|-|S|-1)!}{|D|!} (\mathcal{G}(S \cup \{p\}) - \mathcal{G}(S)). \tag{6}$$

An intuitive explanation of this formula is as follows. Consider the situation when we form the coalition D by selecting random players from D, one by one, randomly and uniformly (without replacement). The Shapley value of p is the expected (or average) contribution of p when it is added to the set of players selected up to that point. Put differently, when ordering the players of D in a random permutation, how does $\mathcal{G}$ differ between the prefix that precedes p and the one that ends with p? Accordingly, $|S|!(|D| - |S| - 1)!$ corresponds to the number of permutations of D with all players in S coming first, then p, and then all the others.

In our case where the players are the tuples in the database D, we view D as consisting of two parts, D_{x} and D_{n}; the former comprises the *exogenous* tuples and the latter the *endogenous* tuples. A player p of the general setting becomes an endogenous tuple $\tau \in D_{\mathsf{n}}$, and, in Formula (6), S and $S \cup \{\tau\}$ become subinstances of D_{n}. The wealth function

$\mathcal{G}(S)$ becomes $\mathcal{Q}[D_x \cup S] - \mathcal{Q}[D_x]$. Note that $\mathcal{Q}[D_x]$ is the result of the query on the subinstance that contains all exogenous and none of the endogenous tuples. The subtraction of $\mathcal{Q}[D_x]$ is due to the formal requirement of the wealth function underlying the Shapley value to be zero on the empty set of players. The Shapley value of a tuple then becomes:

$$\mathsf{Shapley}_{\mathcal{Q}}(D,\tau) \stackrel{\text{def}}{=\!=} \sum_{S \subseteq D_n \setminus \{\tau\}} \frac{|S|!(|D_n| - |S| - 1)!}{|D_n|!} (\mathcal{Q}[D_x \cup S \cup \{\tau\}] - \mathcal{Q}[D_x \cup S]) \,. \tag{7}$$

Example 5. Consider a very simple database instance D represented by the following table.

Players	name	amount
	john	5
	joe	2
τ	**sue**	**4**
	mary	5
	peggy	14

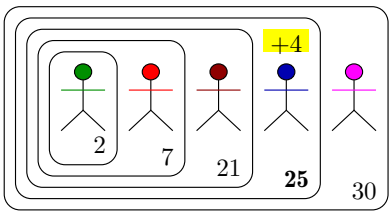
Here we assume that all tuples are endogenous, that is, $D_n = D$ and $D_x = \emptyset$. The query

$$\mathcal{Q}\colon \ sum\{\!\{y \mid \exists x Players(x,y)\}\!\}$$

represents the sum of numbers in the last attribute. We wish to quantify, via the Shapley value, the contribution of the selected tuple τ, namely $Players(sue, 4)$, to the answer.

The table above does not encode any particular order among the tuples. One particular subset S of $D \setminus \{\tau\}$ contains the tuples indicated by an arrow in the table below. The second to last tuple is not chosen. One possible permutation of D, and also of S, is given by the auxiliary tuple numbers on the left-hand side. It is also shown in the figure. The numbers at the lower-right corners show the total wealth of the players in the game so far. With this particular permutation, the contribution of τ, namely $(\mathcal{Q}[S \cup \{\tau\}] - \mathcal{Q}[S])$ in the sum in (7), is 4.

Players	name	amount	
#2	john	5	←
#1	joe	2	←
#4	**sue**	**4**	τ
#5	mary	5	×
#3	peggy	14	←



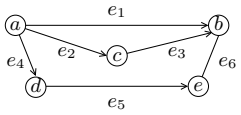
This contribution of 4 will appear in the sum as many times as the number of permutations of S, that is, 6 times. If we had not only tuple #5, but also a sixth one, say #6, outside S, we would multiply 4×6 by 2 (for the two possible permutations of the tuples left outside S). □

It is well-known that, in various instantiations of coalition games, the computation of the Shapley value for a player can be hard to compute, actually #P-hard. This means it is at least as difficult as any problem in the class #P that contains the computational problems of counting the solutions of problems in the class NP, that is, decision problems that can be solved in polynomial time by means of a nondeterministic Turing machine [3]. Among the best-known and hard problems in the class #P, we find #SAT, i.e., the problem of computing the number of satisfying truth valuations for a propositional formula. It should be clear that this problem is at least as hard as SAT, since a formula is satisfiable if and only if the number of solutions is nonzero.

3. COMPARING THE MEASURES

We now illustrate the differences among the contribution measures we mentioned in the previous sections, namely *responsibility*, *causal-effect*, and *Shapley value*, on several examples. We begin with the following example, taken from Salimi et al. [31], that was used as a motivation to introduce an alternative to the notion of causal responsibility, that of causal-effect.

Example 6. Consider a database D defined via a table *Edge* with two attributes. Each tuple in the table represents an edge of the following graph.



Here, we assume that all edges e_i are endogenous tuples. Let $\mathcal{Q}_{ab}$ be the Boolean query (definable in, e.g., Datalog, or as a UCQ) that determines whether there is a path from a to b. Let us compute the contribution of different edges e_i to the query result. Intuitively, we expect e_1 to have the highest value as it provides a direct path from a to b, while e_2 contributes to a path only in the presence of e_3, and e_4 enables a path only in the presence of both e_5 and e_6.

All tuples in D are actual causes since every tuple appears in a path from a to b. It is easy to verify that all the tuples in D have the same causal responsibility, $\frac{1}{3}$, which may be considered as counter-intuitive. In that sense, the causal-effect and Shapley value return more intuitive results, as we have that [31]:

$$\mathsf{CE}_{\mathcal{Q}_{ab}}(D, e_1) = 0.65625$$
$$\mathsf{CE}_{\mathcal{Q}_{ab}}(D, e_i) = 0.21875, \text{ for } i \in \{2,3\}$$
$$\mathsf{CE}_{\mathcal{Q}_{ab}}(D, e_i) = 0.09375, \text{ for } i \in \{4,5,6\}$$

where we use the notation $\mathsf{CE}_{\mathcal{Q}_{ab}}(D, e_i)$ for the causal effect of $e_i \in D$ w.r.t. $\mathcal{Q}_{ab}$, and:

$$\mathsf{Shapley}_{\mathcal{Q}_{ab}}(D, e_1) = 0.5833$$
$$\mathsf{Shapley}_{\mathcal{Q}_{ab}}(D, e_i) = 0.1333, \text{ for } i \in \{2,3\}$$
$$\mathsf{Shapley}_{\mathcal{Q}_{ab}}(D, e_i) = 0.05, \text{ for } i \in \{4,5,6\}$$

(The detailed computations can be found in [23].) □

Note that the responsibility measure is fundamentally designed for non-numerical queries, and it is not at all clear whether it can incorporate the numerical contribution of a tuple (e.g., recognizing that some tuples contribute more than others due to high numerical attributes). Therefore, in the following example, we only consider the causal-effect measure and the Shapley value.

Example 7. Consider again the query $\mathcal{Q}_3$ in (3), and assume that all the tuples in the table *Receives'* in Example 3 are exogenous, while the tuples in the table *Store* are endogenous. It is rather straightforward to see that:

$$\mathsf{CE}_{\mathcal{Q}_3}(D, \mathit{Store}(s_2)) = \mathsf{Shapley}_{\mathcal{Q}_3}(D, \mathit{Store}(s_2)) = 0$$

as this tuple has not impact on the query result (that is, the addition of this tuple to any subset of the endogenous tuples does not change the query result).

For the other two tuples of *Store*, a simple computation shows that:

$$\mathsf{CE}_{\mathcal{Q}_3}(D, \mathit{Store}(s_3)) = \mathsf{Shapley}_{\mathcal{Q}_3}(D, \mathit{Store}(s_3)) = 11$$

$$\mathsf{CE}_{\mathcal{Q}_3}(D, \mathit{Store}(s_4)) = \mathsf{Shapley}_{\mathcal{Q}_3}(D, \mathit{Store}(s_4)) = 9. \qquad \square$$

These two examples show that the numbers for the causal-effect measure and the Shapley value may coincide in some cases, and be different in other cases. In both examples, however, the two measures rank the tuples similarly according to their contribution to the query result.

We showed in [23] that the causal-effect score coincides with another popular score used in cooperative game theory and related areas, namely the *Banzhaf Power Index* [16]. Its definition is similar to that of the Shapley value, but instead of considering permutations of subsets of players, only subsets of players are considered, which has advantages in some applications. In general, its computation is also intractable [17]. It is known that the Shapley value and Banzhaf power index may produce different rankings in general [30].

While the justification to measuring tuple contribution using one measure over the other is yet to be established, we believe that the suitability of the Shapley value is backed by the aforementioned theoretical justification as well as its massive adoption in a plethora of fields.

4. COMPUTATIONAL COMPLEXITY

We now discuss the computational complexity of calculating the Shapley value of a database tuple with respect to a database query, either exactly or approximately.

4.1 Conjunctive Queries

We first discuss the case of *Boolean* CQs, that is, CQs such as $\mathcal{Q}$ of Equation (1) where all variables are existentially quantified. For the fragment of CQs without self-joins (i.e., no relation name is mentioned more than once), we can classify all the queries into tractable (polynomial-time) and intractable (#P-hard) ones. Interestingly, the tractability criterion is the same as that of query answering over tuple-independent probabilistic databases [13]: being *hierarchical*. (We recall the precise definition later on.)

THEOREM 4.1. *Let $\mathcal{Q}$ be a Boolean CQ without self-joins. If $\mathcal{Q}$ is hierarchical, then $\mathsf{Shapley}_{\mathcal{Q}}(D, \tau)$ can be computed in polynomial time, given D and τ. Otherwise, the problem is #P-hard.*

Recall that #P is the complexity class of problems that can be defined as that of counting the witnesses of a problem in NP. A Boolean CQ $\mathcal{Q}$ is *hierarchical* if for every two variables x and y it holds that the set of atoms (conjuncts) that contain x either contains, is contained in, or is disjoint from the set of atoms that contain y. For example, every CQ with at most two atoms (e.g., the query $\mathcal{Q}_1'$ in (4)) is hierarchical. The query $\mathcal{Q}_3$ of Example 4, on the other hand, is not hierarchical, since the sets of atoms that contain the variables x and y are $\{\mathit{IntStore}(x), \mathit{Receives}'(x, y, z)\}$ and $\{\mathit{Receives}'(x, y, z), \mathit{Store}(y)\}$, respectively, which have neither disjointness nor containment between them.

For illustration, we conclude from Theorem 4.1 that the Shapley value is tractable for $\mathcal{Q}_1'$ of Equation (4), but intractable for $\mathcal{Q}_3$ of Equation (5). On the other hand, the theorem does not say anything about the complexity of $\mathcal{Q}$ in Equation (1), since $\mathcal{Q}$ has a self-join (as it contains two occurrences of the *Store* relation). Nevertheless, an easy reduction from the case of $\mathcal{Q}_3$ shows that the Shapley value is #P-hard to compute for $\mathcal{Q}$ as well (see, e.g., [29]).

The algorithm for hierarchical Boolean CQs $\mathcal{Q}$ without self-joins is a reduction to the following counting problem: given a database D and a number k, how many sets of k tuples from D satisfy $\mathcal{Q}$? (The problem of counting the subsets of D that satisfy $\mathcal{Q}$, regardless of their size, has been the subject of recent studies [1, 22].) In [23] we show that when $\mathcal{Q}$ is a hierarchical CQ without self-joins, this problem is solvable in polynomial time.

Similarly to Dalvi and Suciu [14], our proof of hardness in [23] consists of two steps. First, we prove hardness for the simplest non-hierarchical query

$$\mathcal{Q}_{\mathsf{RST}}\colon \exists x \exists y (R(x) \wedge S(x, y) \wedge T(y))\,.$$

Then, we reduce the computation of $\mathsf{Shapley}_{\mathcal{Q}_{\mathsf{RST}}}(D, \tau)$ to that of $\mathsf{Shapley}_{\mathcal{Q}}(D, \tau)$ for any non-hierarchical CQ $\mathcal{Q}$ without self-joins. The second step is the same as that of Dalvi and Suciu [14]. The proof of the first step is by a reduction from the problem of computing the number of independent sets of a bipartite graph. Our reduction adopts a technique that Aziz and Keijzer [4] used for proving the hardness of computing the Shapley value for a *matching game* on unweighted graphs: solve several instances of the problem in order to construct a full-rank set of linear equations.

To generalize our complexity results to non-Boolean CQs, we apply the aforementioned standard approach of converting the CQ into a Boolean CQ by referring to each output variable as a constant. The idea is that we view every answer as a separate Boolean CQ. (Recall that we are using data complexity.) Hence, for a CQ $\mathcal{Q}$ we consider the Boolean version $\mathcal{Q}_b$ where each free (output) variable becomes a constant. When $\mathcal{Q}_b$ is a Boolean CQ without self-joins, the complexity of calculating the Shapley value of a tuple τ for any answer to $\mathcal{Q}$ is the same as that of calculating $\mathsf{Shapley}_{\mathcal{Q}_b}(D, \tau)$. In particular, we can compute it in polynomial time if $\mathcal{Q}_b$ is hierarchical, and it is #P-hard otherwise. For example, in the following variation of $\mathcal{Q}_3$ in (5), we can efficiently compute the Shapley value of every tuple to every query answer.

$$\mathcal{Q}'(x)\colon\ \exists y \exists z (\mathit{IntStore}(x) \wedge \mathit{Receives}'(x, y, z) \wedge \mathit{Store}(y))$$

This is because the Boolean CQ

$$\mathcal{Q}_b'\colon\ \exists y \exists z (\mathit{IntStore}(s_2) \wedge \mathit{Receives}'(s_2, y, z) \wedge \mathit{Store}(y))\,.$$

is a hierarchical CQ.

4.2 Aggregate Queries

Next, we consider aggregate queries. For simplicity of presentation, the aggregate queries that we consider here have the form $\mathcal{Q}(D) = \alpha\{\!\{x \mid \mathcal{Q}'(D)\}\!\}$ when applied to a database D. Here, $\mathcal{Q}'$ is a CQ, x is one of the free variables of $\mathcal{Q}'$ that we view as a numerical attribute, and α is an *aggregate*

operator that maps a bag of numbers into a single number. Recall that $\{\!\{\cdot\}\!\}$ is used here for *bag* notation. Examples of α include the functions *sum*, *min*, *max*, *average* and so on. We call $\mathcal{Q}'$ the *underlying CQ* of $\mathcal{Q}$.

REMARK 1. *In the conference version of this article [23], we consider a more general class of queries where, instead of the variable x, we allow for any* feature function *that transforms a given tuple into a number (e.g., the product of two attributes). The results we state here hold for this generalized model as well.*

As expected, the complexity of an aggregate query depends on the complexity of its underlying CQ. The following theorem generalizes the hardness side of Theorem 4.1 and states that it is #P-hard to compute $\mathsf{Shapley}_{\mathcal{Q}}(D,\tau)$ whenever the underlying CQ is a non-hierarchical CQ without self-joins. The only exception is the case where $\mathcal{Q}$ is a *constant* query, that is, $\mathcal{Q}(D) = \mathcal{Q}(D')$ for all databases D and D'—in that case, $\mathsf{Shapley}_{\mathcal{Q}}(D,\tau) = 0$ always holds.

THEOREM 4.2. *Let $\mathcal{Q}$ be a fixed aggregate query where the underlying CQ is a non-hierarchical CQ without self-joins. If $\mathcal{Q}$ is not constant, then computing $\mathsf{Shapley}_{\mathcal{Q}}(D,\tau)$, given D and τ, is* #P*-hard.*

For instance, it follows from Theorem 4.2 that, computing $\mathsf{Shapley}_{\mathcal{Q}}(D,\tau)$ for the query

$$\mathcal{Q}_4\colon\ \alpha\{\!\{z \mid (\mathit{IntStore}(x) \wedge \mathit{Receives}'(x,y,z) \wedge \mathit{Store}(y))\}\!\} \quad (8)$$

is hard for all $\alpha \in \{\mathit{min}, \mathit{max}, \mathit{sum}, \mathit{average}\}$, and, in fact, for any aggregate function α that is not a constant.

What about the other direction? Does the tractability of the Shapley value for the underlying CQ imply the tractability of the Shapley value for the whole aggregate query? We do not have any result that is as general as that of Theorem 4.2, but we can show that this is the case for the aggregate operator *sum* (and *count* as a special case). This is a simple corollary of Theorem 4.1 that we obtain by applying the linearity of expectation.

COROLLARY 4.3. *Let $\mathcal{Q}(D) = \mathit{sum}\{\!\{x \mid \mathcal{Q}'(D)\}\!\}$. If $\mathcal{Q}'$ is a hierarchical CQ without self-joins, then $\mathsf{Shapley}_{\mathcal{Q}}(D,\tau)$ can be computed in polynomial time, given D and τ.*

As an example, the Shapley value can be computed in polynomial time for the query $\mathcal{Q}_1$ of Example 3, because the underlying CQ has only two atoms, and is then hierarchical.

The complexity of computing the Shapley value for other aggregate queries remains an open problem for the general case where the underlying CQ is a hierarchical CQ without self-joins. We can, however, state a positive result for *max* and *min*, for the special case where the underlying CQ consists of a single atom (i.e., aggregation over a single relation). As an example, computing $\mathsf{Shapley}_{\mathcal{Q}}(D,\tau)$ can be done in polynomial time for:

$$\mathcal{Q}\colon \{\!\{\mathit{max}\{z \mid \exists y(\mathit{Receives}'(s_2,y,z))\}\!\}$$

However, we do not know whether this is also the case for the query $\mathcal{Q}_2$ of Example 3.

4.3 Approximation

The complexity results presented so far imply that computing the exact Shapley value is often intractable. Nevertheless, the picture is far more optimistic when allowing approximation with strong precision guarantees. A conventional feasibility notion of arbitrarily-tight approximations is via efficient approximation *schemes* such as the *Fully Polynomial-Time Approximation Scheme* (*FPRAS* for short). An FPRAS for a numeric function f is a randomized algorithm $A(x,\epsilon,\delta)$, where x is an input for f and $\epsilon,\delta \in (0,1)$, that returns an ϵ-approximation of $f(x)$ with probability $1-\delta$ (where the probability is over the randomness of A) in time polynomial in x, $1/\epsilon$ and $\log(1/\delta)$. To be more precise, we distinguish between an *additive* (or *absolute*) FPRAS:

$$\Pr\left[f(x) - \epsilon \le A(x,\epsilon,\delta) \le f(x) + \epsilon\right] \ \ge\ 1-\delta$$

and a *multiplicative* (or *relative*) FPRAS:

$$\Pr\left[\frac{f(x)}{1+\epsilon} \le A(x,\epsilon,\delta) \le (1+\epsilon)f(x)\right] \ \ge\ 1-\delta\,.$$

Using the Chernoff-Hoeffding bound, we easily get an additive FPRAS of $\mathsf{Shapley}_{\mathcal{Q}}(D,\tau)$ when $\mathcal{Q}$ is *any* monotone Boolean query computable in polynomial time, by simply taking the ratio of successes over $O(\log(1/\delta)/\epsilon^2)$ trials of the following experiment:

1. Select a random permutation $(t_1,\ldots,t_n)$ over the set of endogenous tuples in the database,
2. Suppose that $\tau = t_i$, and let $D_{i-1} = D_{\mathsf{x}} \cup \{t_1,\ldots,t_{i-1}\}$. If $\mathcal{Q}(D_{i-1})$ is false and $\mathcal{Q}(D_{i-1} \cup \{\tau\})$ is true, then report "success;" otherwise, "failure."

Moreover, the additive FPRAS extends to non-Boolean CQs in the same manner described in Section 4.1.

In general, the existence of an additive FPRAS for a function f does not guarantee the existence of a multiplicative one, since $f(x)$ can be very small. For example, we can get an additive FPRAS of the satisfaction of a propositional formula over Boolean i.i.d. variables by, again, sampling the averaging, but there is no multiplicative FPRAS for such formulas unless NP $\subseteq$ BPP. Nevertheless, the situation is different for $\mathsf{Shapley}_{\mathcal{Q}}(D,\tau)$ when $\mathcal{Q}$ is a CQ, and even a UCQ, since the Shapley value is *never* too small. In particular, we have the following "gap" property.

PROPOSITION 4.4. *Let $\mathcal{Q}$ be a fixed Boolean UCQ. Then $\mathcal{Q}$ satisfies the* gap property*: there is a polynomial p such that for all databases D and tuples τ of D it is the case that $\mathsf{Shapley}_{\mathcal{Q}}(D,\tau)$ is either zero or at least $1/(p(|D|))$.*

It follows from Proposition 4.4 that a multiplicative FPRAS can be obtained using the above sampling algorithm, possibly with a different (yet still polynomial) number of samples. Hence, we have the following.

COROLLARY 4.5. *For every fixed UCQ, the Shapley value has both an additive and a multiplicative FPRAS.*

Observe that Corollary 4.5 does not make any assumption about self-joins—it allows for any UCQ. As we explain shortly, it ceases to hold once *negation* is allowed. Corollary 4.5 also generalizes to a multiplicative FPRAS for summation over CQs, in the case where all the values in the summation have the same sign (i.e., they are either all negative or all non-negative). In other words, there is a multiplicative FPRAS for the query $\mathcal{Q}(D) = \mathit{sum}\{\!\{x \mid \mathcal{Q}'(D)\}\!\}$ under the assumption that all of the values that x is assigned have the same sign.

What changes when we allow for *negation*? This problem has been studied by Reshef et al. [29]. On the positive side, the additive approximation is tractable even when UCQs are allowed to use (safe) negation.

PROPOSITION 4.6. ([29]) *For every fixed UCQ with negation, the Shapley value has an additive FPRAS.*

Yet, the "gap" property can be violated when CQs are allowed to use negation. Reshef et al. [29] have shown that we lose this property when considering CQs with negated atoms. For example, when considering the query

$$\exists x \exists y (Store(x) \wedge Receives(x, y) \wedge \neg Store(y))$$

that differs from the query $\mathcal{Q}$ of Example 1 only by the negation of the last atom, the Shapley value of a tuple may be as small as $2^{-\Theta(|D|)}$. In fact, a CQ with negation almost always violates the gap property [29, Theorem 5.1].

While the gap property is a technique for extending additive approximations into multiplicative ones, its violation does not preclude the existence of *any* multiplicative approximation. Nevertheless, Reshef et al. [29] have shown that there are CQs with negation where a multiplicative approximation is infeasible, since it is already NP-hard to determine whether the Shapley value is nonzero. An example is the following CQ:

$$\exists x \exists y \exists z \exists w (T(z) \wedge \neg R(x) \wedge \neg R(y) \wedge R(z) \wedge R(w) \wedge S(x, y, z, w)).$$

5. CONCLUSIONS

Explanations in AI, Machine Learning, in particular, and Data and Knowledge Management have become prominent and active areas of research. There are different notions about, and approaches to, explanations in those fields (and more generally and traditionally, in Science and Philosophy). There is still much debate about what is an explanation, and even more, about what is a *good explanation*.

Many approaches in the broad areas we just mentioned are based on some sort of causal analysis, and more specifically, on counterfactual analysis. We could say that the Shapley value has a rather implicit counterfactual component, through the average of the game outcome depending on the presence or not of a particular player, under different scenarios related to the other players.

Explanations are usually, but not always, expressed as numerical scores that represent the importance of a player, a database tuple, a feature value (c.f. below), etc., for the outcome of a function, a database, or a computational model. In this work, we have adopted this approach to the explanations for query answers from relational databases.

In Section 1, we mentioned the *causal-effect* [31] as an explanation score for database tuples and query answering, as an alternative to *responsibility*. All of our complexity results for the exact computation of the Shapley value are also applicable to the causal-effect measure (and Banzhaf power index). However, we can show that the "gap" property does not hold for this measure, and the question of whether there exists a multiplicative approximation remains open for future investigation.

The Shapley value has been used in a similar yet different manner for another fundamental task in data management: measuring the responsibility of a tuple to the level of *inconsistency* [28, 36]. For such a measure to be applicable, one needs first to adopt a measure for quantifying the amount of inconsistency of the database. More concretely, such a measure should indicate the extent to which the database's integrity constraints are violated. In a more recent work [24], we have studied the complexity of the Shapley value of tuples under various natural measures of inconsistency: the number of violations, the number of tuples that participate in violations, the number of repairs, and the minimal number of tuples that one needs to delete to retain consistency [5].

It is worth mentioning, for a broader view of things, that the Shapley value has been used lately in machine learning, to quantify the relevance of a feature value for the outcome of a model (e.g., the result of a classifier), which is important in *Explainable AI* (XAI). In this area, it is known as the *SHAP-score* [25]; and has been applied both with black-box and open-box models. Only the input/output relation is needed for its computation. However, the availability of the model may make the computation much more efficient [2,37], actually tractable in some cases. In [7], the SHAP-score was experimentally compared with other scores, such as, *RESP*, an adaptation of the responsibility score to the classification setting, and the Rudin-score for FICO data [11].

There are many directions in which our and related work could be extended. One that looks particularly promising and relevant has to do with the definition or computation of explanation scores in combination with domain knowledge or, *semantics*, in more general terms [6]. In particular, this additional knowledge could inform the scores about the explanations, such as database tuples, feature values, etc., that become useful, or more technically, *actionable* or *algorithmic recourses* [21], i.e. something we can do something with. For example, if a loan application is rejected due to blurry personal payment history, we might be in a position to clarify our records. However, if it is due to our low educational level, it might be too late (or impossible) to do anything about it.

Acknowledgments

The work of L. Bertossi was funded by ANID - Millennium Science Initiative Program- Code ICN17002. The work of Ester Livshits, Benny Kimelfeld, and Moshe Sebag was supported by the Israel Science Foundation (ISF), grants 1295/15 and 768/19, and the Deutsche Forschungsgemeinschaft (DFG) project 412400621 (DIP program). The work of Ester Livshits was also supported by the Technion Hiroshi Fujiwara Cyber Security Research Center and the Israel Cyber Bureau.

6. REFERENCES

[1] A. Amarilli and B. Kimelfeld. Model counting for conjunctive queries without self-joins. *CoRR*, abs/1908.07093, 2019. To appear at ICDT 2021.

[2] M. Arenas, P. Barceló, L. Bertossi, and M. Monet. The tractability of SHAP-scores over deterministic and decomposable boolean circuits. In *Proceedings of AAAI*, 2021. CoRR abs/2007.14045.

[3] S. Arora and B. Barak. *Computational Complexity - A Modern Approach*. Cambridge University Press, 2009.

[4] H. Aziz and B. de Keijzer. Shapley meets Shapley. In *STACS*, pages 99–111, 2014.

[5] L. Bertossi. Repair-based degrees of database inconsistency. In *LPNMR*, volume 11481 of *LMCS*, pages 195–209. Springer, 2019.

[6] L. Bertossi. Declarative approaches to counterfactual explanations for classification. *CoRR*, abs/2011.07423, 2020. Extended version of RuleML+RR'20 paper.

[7] L. Bertossi, J. Li, M. Schleich, D. Suciu, and Z. Vagena. Causality-based explanation of classification outcomes. In *DEEM@SIGMOD*, pages 6:1–6:10. ACM, 2020.

[8] L. Bertossi and B. Salimi. Causes for query answers from databases: Datalog abduction, view-updates, and integrity constraints. *Int. J. Approx. Reason.*, 90:226–252, 2017.

[9] L. Bertossi and B. Salimi. From causes for database queries to repairs and model-based diagnosis and back. *Theory Comput. Syst.*, 61(1):191–232, 2017.

[10] P. Buneman, S. Khanna, and W. C. Tan. Why and where: A characterization of data provenance. In *ICDT*, volume 1973 of *Lecture Notes in Computer Science*, pages 316–330. Springer, 2001.

[11] C. Chen, K. Lin, C. Rudin, Y. Shaposhnik, S. Wang, and T. Wang. An interpretable model with globally consistent explanations for credit risk. *CoRR*, abs/1811.12615, 2018.

[12] H. Chockler and J. Y. Halpern. Responsibility and blame: A structural-model approach. *J. Artif. Intell. Res.*, 22:93–115, 2004.

[13] N. N. Dalvi, C. Ré, and D. Suciu. Probabilistic databases: diamonds in the dirt. *Commun. ACM*, 52(7):86–94, 2009.

[14] N. N. Dalvi and D. Suciu. Efficient query evaluation on probabilistic databases. In *VLDB*, pages 864–875. Morgan Kaufmann, 2004.

[15] N. N. Dalvi and D. Suciu. The dichotomy of probabilistic inference for unions of conjunctive queries. *J. ACM*, 59(6):30:1–30:87, 2012.

[16] P. Dubey and L. S. Shapley. Mathematical properties of the Banzhaf power index. *Mathematics of Operations Research*, 4(2):99–131, 1979.

[17] G. Greco, F. Lupia, and F. Scarcello. Structural tractability of Shapley and Banzhaf values in allocation games. In *IJCAI*, pages 547–553, 2015.

[18] T. J. Green and V. Tannen. The semiring framework for database provenance. In *PODS*, pages 93–99. ACM, 2017.

[19] J. Y. Halpern. A modification of the Halpern-Pearl definition of causality. In *IJCAI*, pages 3022–3033. AAAI Press, 2015.

[20] J. Y. Halpern and J. Pearl. Causes and Explanations: A Structural-Model Approach. Part I: Causes. *The British Journal for the Philosophy of Science*, 56(4):843–887, 2005.

[21] A. Karimi, B. J. von Kügelgen, B. Schölkopf, and I. Valera. Algorithmic recourse under imperfect causal knowledge: a probabilistic approach. In *NeurIPS*, 2020.

[22] B. Kenig and D. Suciu. A dichotomy for the generalized model counting problem for unions of conjunctive queries. *CoRR*, abs/2008.00896, 2020.

[23] E. Livshits, L. Bertossi, B. Kimelfeld, and M. Sebag. The shapley value of tuples in query answering. In *ICDT*, volume 155 of *LIPIcs*, pages 20:1–20:19, 2020.

[24] E. Livshits and B. Kimelfeld. The shapley value of inconsistency measures for functional dependencies. *CoRR*, abs/2009.13819, 2020. To appear at ICDT 2021.

[25] S. M. Lundberg, G. Erion, H. Chen, A. D. Grave, J. Prutkin, B. Nair, R. Katz, J. Himmelfarb, N. Bansal, and S.-I. Lee. From local explanations to global understanding with explainable AI for trees. *Nature Machine Intelligence*, 2(1):56–67, 2020.

[26] A. Meliou, W. Gatterbauer, J. Y. Halpern, C. Koch, K. F. Moore, and D. Suciu. Causality in databases. *IEEE Data Eng. Bull.*, 33(3):59–67, 2010.

[27] A. Meliou, W. Gatterbauer, K. F. Moore, and D. Suciu. The complexity of causality and responsibility for query answers and non-answers. *Proc. VLDB Endow.*, 4(1):34–45, 2010.

[28] K. Mu, W. Liu, and Z. Jin. Measuring the blame of each formula for inconsistent prioritized knowledge bases. *Journal of Logic and Computation*, 22(3):481–516, 02 2011.

[29] A. Reshef, B. Kimelfeld, and E. Livshits. The impact of negation on the complexity of the shapley value in conjunctive queries. In *PODS*, pages 285–297. ACM, 2020.

[30] D. G. Saari and K. K. Sieberg. Some surprising properties of power indices. *Games Econ. Behav.*, 36(2):241–263, 2001.

[31] B. Salimi, L. Bertossi, D. Suciu, and G. V. den Broeck. Quantifying causal effects on query answering in databases. In *TaPP*. USENIX Association, 2016.

[32] L. S. Shapley. *A Value for n-Person Games*. RAND Corporation, Santa Monica, CA, 1952.

[33] L. S. Shapley and A. E. Roth. *The Shapley value : essays in honor of Lloyd S. Shapley*. Cambridge, 1988.

[34] P. Struss. Model-based problem solving. In *Handbook of Knowledge Representation*, volume 3 of *Foundations of Artificial Intelligence*, pages 395–465. Elsevier, 2008.

[35] D. Suciu, D. Olteanu, C. Ré, and C. Koch. *Probabilistic Databases*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2011.

[36] M. Thimm. Measuring inconsistency in probabilistic knowledge bases. In *UAI*, pages 530–537, 2009.

[37] G. Van den Broeck, A. Lykov, M. Schleich, and D. Suciu. On the tractability of SHAP explanations. In *Proceedings of AAAI*, 2021. CoRR abs/2009.08634.

Technical Perspective: Scaling Dynamic Hash Tables on Real Persistent Memory

Kenneth A. Ross
Columbia University
kar@cs.columbia.edu

Byte-addressable persistent memory was considered in the data management community as long ago as 1986. Thatte saw the advantages for programmability in unifying the abstractions of byte-addressable RAM with persistence [2]. Thatte's context was object-oriented databases containing a variety data structures that would be awkward to transform into the block-oriented abstractions provided by typical secondary storage. Thatte's proposed physical instantiation of persistent memory was a disk-backed device, although it is unclear whether such a device was ever built. Thatte recognized the importance of recovery to the overall scheme.

Fast forward to 2017 when Intel released its Optane nonvolatile memory. Among the key benefits of Optane is byte-addressibility, making it a strong candidate for use as a persistent memory. There are many different aspects of Optane memory that might influence how data structures behave when implemented on Optane rather than on RAM. Traditional measures such as latency and throughput (which may differ for reads and writes) and capacity are important. Particular use cases may depend on other measures, such as the supported concurrency level and the speed of atomic operations. Some details of the device, such as the internal block size (analogous to the cache line size for RAM or the block size for a secondary storage device) may affect performance without being fundamental to the technology. Given that all of these measures are quite different from the corresponding parameters for RAM and for SSDs, the ideal data structure configurations for such devices may look quite different from those used before.

In this context, Lu et al. [1] have tackled the question of how best to design a persistent dynamic hash table using persistent memory, evaluating their design on the Optane platform. Persistent dynamic hash tables have broad applicability for storing data and indices in data management and other applications, and therefore constitute a workload worth optimizing. Prior work in this area included designs that were proposed before widespread availability of Optane devices. As Lu et al. [1] demonstrate experimentally, there were several "gotchas" in those implementations that caused poor performance on Optane because they accessed the device in a manner that turned out to be inefficient. Informed by the Optane device, Lu et al. [1] have engineered a highly performant implementation of persistent hash tables.

What makes this work stand out is the way many competing demands of the data structure are addressed in a balanced way. Unlike traditional hash tables, load factors are kept high by allowing multiple possible destinations for a key. Variable-length keys are supported. Failed searches are optimized by maintaining metadata that can short-circuit such look-ups. High concurrency is achieved through careful implementation of optimistic synchronization primitives. Fast recoverability is based on a timestamp-based scheme that allows the system to be available in constant time, with recovery work leading to somewhat degraded performance for a short period after recovery. The implementation is open-source, allowing others to build on these insights.

While the achievements of Lu et al. [1] are impressive, their paper highlights several issues for future research. It is not easy to program efficient data structures for devices like Optane, despite the byte-addressable interface. Complex issues such as leaks of persistent memory are more serious than leaks in RAM. Some of the choices made by Lu et al. [1] depend on knowing the internal block size of the device, something that may be different in future devices, and may vary across devices. Some kind of self-tuning might be needed to make the right choices based on measured performance rather than inside knowledge of such parameters.

The future role of persistent memory in database management systems remains open. Simply switching the underlying memory type to Optane without changing the data structures is not efficient [3]. For inner-loop data structures like hash tables, it may be worth the effort to write specialized code so that the advantages of persistent memory translate into tangible benefits for database users.

1. REFERENCES

[1] B. Lu, X. Hao, T. Wang, and E. Lo. Dash: Scalable hashing on persistent memory. *Proc. VLDB Endow.*, 13(8):1147–1161, 2020.

[2] S. M. Thatte. Persistent memory: A storage architecture for object-oriented database systems. In *Proceedings on the 1986 International Workshop on Object-Oriented Database Systems*, pages 148–159. IEEE Computer Society Press, 1986.

[3] Y. Wu, K. Park, R. Sen, B. Kroth, and J. Do. Lessons learned from the early performance evaluation of Intel Optane DC persistent memory in DBMS. In *Proceedings of the 16th International Workshop on Data Management on New Hardware*, DaMoN '20, 2020.

Scaling Dynamic Hash Tables on Real Persistent Memory

Baotong Lu[1]* Xiangpeng Hao[2] Tianzheng Wang[2] Eric Lo[1]
[1]The Chinese University of Hong Kong [2]Simon Fraser University
{btlu, ericlo}@cse.cuhk.edu.hk {xha62, tzwang}@sfu.ca

ABSTRACT

Byte-addressable persistent memory (PM) brings hash tables the potential of low latency, cheap persistence and instant recovery. The recent advent of Intel Optane DC Persistent Memory Modules (DCPMM) further accelerates this trend. Many new hash table designs have been proposed, but most of them were based on emulation and perform sub-optimally on real PM. They were also piecewise and partial solutions that side-stepped many important properties, in particular good scalability, high load factor and instant recovery.

We present Dash, a holistic approach to building dynamic and scalable hash tables on real PM hardware with all the aforementioned properties. Based on Dash, we adapted two popular dynamic hashing schemes (extendible hashing and linear hashing). On a 24-core server with Optane DCPMM, compared to state-of-the-art, Dash can achieve up to ∼3.9× higher performance with up to over 90% load factor and an instant recovery time of 57ms regardless of data size.

1. INTRODUCTION

Dynamic hash tables that can grow and shrink as needed at runtime are a fundamental building block of many data-intensive systems, such as database systems [11, 14] and key-value stores [3, 15]. Persistent memory (PM) such as 3D XPoint [1] promises byte-addressability, persistence, high capacity, low cost and high performance, all on the memory bus. These features make PM very attractive for building dynamic hash tables that persist and operate directly on PM, with high performance and instant recovery. The recent release of Intel Optane DC Persistent Memory Module (DCPMM) brings this vision closer to reality. Since PM exhibits several distinct properties (e.g., asymmetric read/write speeds and higher latency); blindly applying prior disk or DRAM based approaches [2, 7] would not reap its full benefits, necessitating a departure from conventional designs.

 This is a minor revision of the paper entitled "Dash: Scalable Hashing on Persistent Memory", published in PVLDB, Vol. 13, No. 8, ISSN 2150-8097. DOI: https://doi.org/10.14778/3389133.3389134
*Work partially performed while at Simon Fraser University.

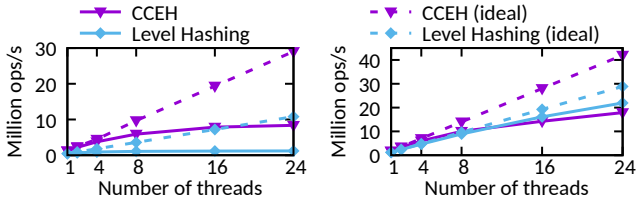


Figure 1: Throughput of state-of-the-art PM hash tables (CCEH [10] and Level Hashing [22]) for insert (left) and search (right) operations on Optane DCPMM. Neither matches the expected scalability.

1.1 Hashing on PM: Not What You Assumed!

There have been a new breed of hash tables specifically designed for PM [10, 17, 22] based on DRAM emulation, before actual PM was available. Their main focus is to reduce cacheline flushes and PM writes for scalable performance. But when they are deployed on real Optane DCPMM, we find (1) scalability is still a major issue, and (2) desirable properties are often traded off.

Figure 1 shows the throughput of two state-of-the-art PM hash tables [10, 22] under insert (left) and search (right) operations, on a 24-core server with Optane DCPMM running workloads under uniform key distribution (details in Section 6). As core count increases, neither scheme scales for inserts, or even read-only search operations. Corroborating with recent work [8, 20], we find the main culprit is Optane DCPMM's limited bandwidth, which is ∼3–14× lower than DRAM's. Although the server is fully populated to provide the maximum possible bandwidth, excessive PM accesses can still easily saturate and prevent the system from scaling. We describe two main sources of excessive PM accesses that were not given enough attention before, followed by a discussion of important but missing functionality in prior work.

Excessive PM Reads. Reducing PM writes has been a main theme in recent work, but many existing solutions incur more PM reads. We note that it is also imperative to reduce PM reads. Different from the device-level behavior (PM reads being faster than writes), *end-to-end* write latency (i.e., the entire data path including CPU caches and write buffers in the memory controller) is often lower than reads [20]. The reason is while PM writes can leverage write buffers, PM reads mostly touch the PM media due to hash table's inherent random access patterns. In particular, existence checks in record probing constitute a large proportion of such PM reads: to find out if a key exists, multiple buckets may

have to be searched, incurring many cache misses and PM reads when comparing keys.

Heavyweight Concurrency Control. Most prior work side-stepped the impact of concurrency control. Bucket-level locking has been widely used [10, 22], but it incurs additional PM writes to acquire/release read locks, further pushing bandwidth consumption towards the limit. Lock-free designs [17] can avoid PM writes for read-only probing operations, but are notoriously hard to get right, more so on PM when safe persistence is necessary [19].

Neither record probing nor concurrency control typically prevents a well-designed hash table to scale on DRAM. However, on PM they can easily exhaust PM's limited bandwidth. These issues call for new designs that can reduce unnecessary PM reads during probing and lightweight concurrency control that further reduces PM writes.

Missing Functionality. We observe in prior designs, necessary functionality was often traded off for performance (though scalability is still an issue on real PM). (1) Indexes could occupy more than 50% of memory capacity [21], so it is critical to improve load factor (records stored vs. hash table capacity). Yet high load factor is often sacrificed by organizing buckets using larger segments in exchange for smaller directories (fewer cache misses) [10]. As we describe later, this in turn can trigger more pre-mature splits and incur even more PM accesses, impacting performance. (2) Variable-length keys are widely used in reality, but prior approaches rarely discuss how to efficiently support them. (3) Instant recovery is a unique, desirable feature that could be provided by PM, but is often omitted in prior work which requires a linear scan of the metadata whose size scales with data size. (4) Prior designs also often side-stepped the PM programming issues (e.g., PM allocation), which impact the proposed solution's scalability and adoption in reality.

1.2 Dash

We present *Dash*, a holistic approach to dynamic and scalable hashing on real PM without trading off desirable properties. Dash uses a combination of new and existing techniques that are carefully engineered to achieve this goal. ❶ We adopt fingerprinting [12] that was used in PM tree structures to avoid unnecessary PM reads during record probing. The idea is to generate fingerprints (one-byte hashes) of keys and place them compactly to summarize the possible existence of keys. This allows a thread to tell if a key possibly exists by scanning the fingerprints which are much smaller than the actual keys. ❷ Instead of traditional bucket-level locking, Dash uses an optimistic, lightweight flavor of it that uses verification to detect conflicts, rather than (expensive) shared locks. This allows Dash to avoid PM writes for search operations. With fingerprinting and optimistic concurrency, Dash avoids both unnecessary reads and writes, saving PM bandwidth and allowing Dash to scale well. ❸ Dash retains desirable properties. We propose a new load balancing strategy to postpone segment splits with improved space utilization. To support instant recovery, we design Dash to only perform a constant amount of work upon restart (reading and possibly writing a one-byte counter) and amortize the "real" recovery work to runtime. Compared to prior work that handles PM programming issues in ad hoc ways, Dash uses well-defined PM programming models (PMDK [4], one of the most popular PM libraries) to systematically handle crash consistency, PM allocation and achieve instant recovery.

Although these techniques are not all new, Dash is the first to integrate them for building hash tables that scale without sacrificing features on real PM. Techniques in Dash can be applied to various static and dynamic hashing schemes. In this paper, we focus on dynamic hashing and apply Dash to two classic and widely-used approaches: extendible hashing [2, 10] and linear hashing [7]. Evaluation using a 24-core Intel Xeon Scalable CPU and 768GB of Optane DCPMM shows that compared to existing state-of-the-art [10, 22], Dash achieves up to ∼3.9× better performance on realistic workloads, up to over 90% of load factor with high space utilization and the ability to instantly recover in 57ms regardless of data size. Our implementation is open-source at: https://github.com/baotonglu/dash.

2. BACKGROUND

We first give necessary background on PM hardware and dynamic hashing, then discuss issues in prior PM hash tables.

2.1 Intel Optane DC Persistent Memory

Hardware. We target Optane DCPMMs (in DIMM form factor). In addition to byte-addressability and persistence, DCPMM offers high capacity at a price lower than DRAM's. Similar to other work [10,22], we leverage its AppDirect mode, as it provides more flexibility and persistence guarantees.

System Architecture. Current mainstream CPU architectures (e.g., Intel Cascade Lake) place DRAM and PM behind multiple levels of *volatile* CPU caches. Data is not guaranteed to be persisted in PM until a cacheline flush instruction (e.g., `CLWB` [5]) is executed or other events that implicitly cause cacheline flush occur. Writes to PM may be reordered, requiring fences to avoid undesirable reordering. The application (hash tables in our case) must issue fences and cacheline flushes to ensure correctness. After a cacheline of data is flushed, it will reach the asynchronous DRAM refresh (ADR) domain which includes a write buffer and a write pending queue with persistence guarantees upon power failure. Once data is in the ADR domain (not necessarily the DCPMM media), it is considered persistent.

Future CPU and DCPMM generations (e.g., the upcoming Intel Ice Lake CPUs) may feature extended ADR (eADR) which further includes the CPU cache in the ADR domain [16], effectively providing persistent CPU caches and thus eliminating the need for cacheline flushes (fences are still needed). The current implementation of Dash still issues cacheline flushes, however, our preliminary experiments that skip cacheline flushes on existing Cascade Lake CPUs showed eADR's potential impact is very small for hash tables. We attribute the reason to hash table's inherently random access patterns.

Performance Characteristics. At the device level, as many previous studies have shown, PM exhibits asymmetric read and write latency, with writes being slower. It exhibits ∼300ns read latency, ∼4× longer than DRAM's. More recent studies [20], however revealed that on Optane DCPMM, read latency *as seen by the software* is often higher than write latency. This is attributed to the fact that writes (`store` instructions) commit (also ensure persistence) once the data reaches the ADR domain at the memory controller rather than when reaching the PM media. On the contrary, a read operation often needs to touch the actual media unless the data being accessed is cache-resident (which is rare in data structures with inherent randomness, e.g., hash tables). Tests also showed that the bandwidth of DCPMM depends

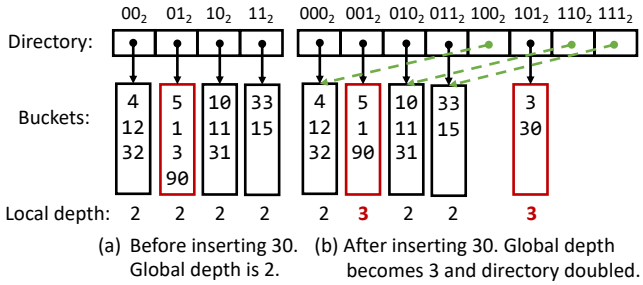


Figure 2: An example of extendible hashing.

on many factors of the workload. In general, compared to DRAM, it exhibits ∼3×/∼8× slower sequential/random read bandwidth. The numbers for sequential/random write are ∼11×/∼14×. Notably, DCPMM exhibits very limited performance for small, random accesses [20], which are inherent access pattern for hash tables. These properties exhibit a stark contrast to prior estimates [13, 18], and lead to significantly lower performance of many prior designs on DCPMM than originally reported. Thus, it is important to reduce *both* PM reads and writes for higher performance.

2.2 Dynamic Hashing

Now we give an overview of extendible hashing [2] and linear hashing [7]. We focus on their memory-friendly versions which PM-adapted hash tables were based upon.

Extendible Hashing. The crux of extendible hashing is its use of a directory to index buckets so that they can be added and removed dynamically at runtime. When a bucket is full, it is split into two new buckets with keys redistributed. The directory may get expanded (doubled) if there is not enough space to store pointers to the new bucket. Figure 2(a) shows an example with four buckets, each of which is pointed to by a directory entry. In the figure, indices of directory entries are shown in binary. The two least significant bits (LSBs) of the hash value are used to select a bucket; we call the number of suffix bits being used here the *global depth*. The hash table can have at most $2^{global\ depth}$ directory entries (buckets). A search operation follows the pointer in the corresponding directory entry to probe the bucket. Each bucket also has a *local depth*. The number of directory entries pointing to one bucket is $2^{global\ depth-local\ depth}$. This allows us to determine whether a directory doubling is needed: if a bucket whose local depth equals the global depth is split (e.g., bucket 01_2 in Figure 2(a)), then the directory needs to be doubled to accommodate the new bucket. Figure 2 shows an example of splitting the full bucket 01_2 when an inserting key 30 is hashed into that bucket. After bucket splitting, *local depth* of the splitting bucket needs to be properly updated. Choosing a proper hash function that evenly distributes keys to all buckets is an important but orthogonal problem.

Linear Hashing. In-memory linear hashing takes a similar approach to organizing buckets using a directory with entries pointing to individual buckets [7]. The main difference compared to extendible hashing is that in linear hashing, the bucket to be split is chosen "linearly." That is, it keeps a pointer (page ID or address) to the bucket to be split next and only that bucket would be split in each round, and advances the pointer to the next bucket when the split of the current bucket is finished. Therefore, the bucket being split is not necessarily the same as the bucket that is full as a result of inserts, and eventually the overflowed bucket will be split and have its keys redistributed. If a bucket is full and an insert is requested to it, more overflow buckets will be created and chained together with the original, full bucket. Compared with extendible hashing, linear hashing could have smaller directory size by proper organization [7].

2.3 Dynamic Hashing on PM

To reduce PM accesses on dynamic extendible hashing, CCEH [10] groups buckets into larger *segments*. Each directory entry then points to a segment which consists of a fixed number of buckets. This design reduces the size of the directory, making it more likely to be cached entirely by the CPU, which helps reducing access to PM. Note that split now happens at the segment (instead of bucket) level. A segment is split once any bucket in it is full, even if the other buckets in the segment still have free slots, which results in low load factor and more PM accesses. To reduce such premature splits, linear probing can be used to allow a record to be inserted into a neighbor bucket. However, this improves load factor at the cost of more PM accesses. Thus, most approaches bound probing distance to a fixed number, e.g., CCEH probes no more than four cachelines. However, our evaluation (Section 6) shows that linear probing alone is not enough in achieving high load factor.

Another important aspect of dynamic PM hashing is to ensure failure atomicity, particularly during segment split which involves lots of PM writes. Existing approaches such as CCEH side-step PM management issues surrounding segment split, having the risk of permanent PM leaks.

3. DESIGN PRINCIPLES

The discussions in Section 2 lead to the following design principles of Dash:

- **Avoid both Unnecessary PM Reads and Writes.** Probing performance impacts not only search operations, but also all the other operations. Therefore, in addition to reducing PM writes, Dash must avoid unnecessary PM reads to conserve bandwidth and alleviate the impact of high end-to-end read latency.
- **Lightweight Concurrency.** Dash must scale well on multicore machines with persistence guarantees. Given the limited bandwidth, concurrency control must be lightweight to incur not much overhead (i.e., avoid PM writes for search operations, such as read locks). Ideally, it should also be relatively easy to implement.
- **Full Functionality.** Dash must not sacrifice or trade off important features that make a hash table useful in practice. In particular, it needs to support instant recovery and variable-length keys and achieve high space utilization.

4. Dash FOR EXTENDIBLE HASHING

Based on the principles in Section 3, we describe Dash in the context of Dash-Extendible Hashing (Dash-EH). We discuss how Dash applies to linear hashing in Section 5.

4.1 Overview

Similar to prior approaches [7, 10], Dash-EH uses segmentation. As shown in Figure 3, each directory entry points to a segment which consists of a fixed number of normal buckets and stash buckets for overflow records from normal buckets

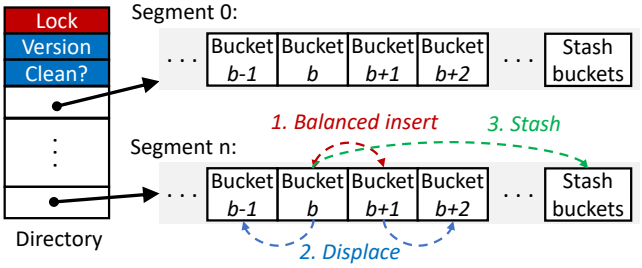


Figure 3: Overall architecture of Dash-EH.

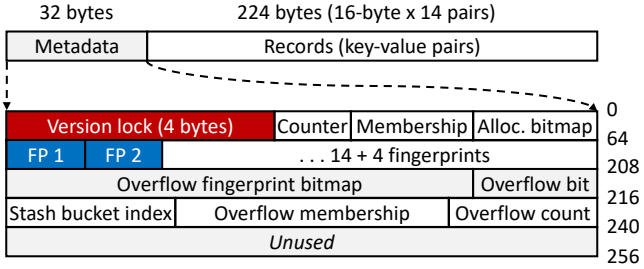


Figure 4: Dash-EH bucket layout. The metadata optimizes probing and load factor, followed by records.

which did not have enough space for the inserts. The lock, version number and clean marker are for concurrency control and recovery, which we describe later.

Figure 4 shows the internals of a bucket. We place the metadata used for bucket probing on the first 32 bytes, followed by multiple 16-byte record slots. The first 8 bytes in each slot store the key (or a pointer to it for keys longer than 8 bytes). The remaining 8 bytes store the payload which is opaque to Dash; it can be an inlined value or a pointer, depending on the application's need. The size of a bucket is adjustable. In our current implementation it is set to 256-byte, which allows us to store 14 records per bucket.

The 32-byte metadata includes key data structures for Dash-EH to handle hash table operations and realize the design principles. It starts with a 4-byte version lock for optimistic concurrency control (Section 4.4). A 4-bit `counter` records the number of records stored in the bucket. The `allocation` bitmap reserves one bit per slot, to indicate whether the corresponding slot stores a valid record. What follows are structures such as fingerprints and overflow metadata to accelerate probing and improve load factor.

4.2 Fingerprinting

Bucket probing (i.e., search in one bucket) is a basic operation needed by all the operations supported by a hash table (search, insert and delete) to check for key existence. Searching a bucket typically requires a linear scan of the slots. This can incur lots of cache misses and is a major source of PM reads, especially so for long keys stored as pointers. It is a major reason for hash tables on PM to exhibit low performance. Moreover, such scans for negative search operations (i.e., when the target key does not exist) are completely unnecessary.

We employ fingerprinting [12] to reduce unnecessary scans. It was used by trees to reduce PM accesses with an amortized number of key loads of one. We adopt it in hash tables to reduce cache misses and accelerate probing. Fingerprints are one-byte hashes of keys for predicting whether a key possibly exists. We use the least significant byte of the key's hash value. To probe for a key, the probing thread first checks whether any fingerprint matches the search key's fingerprint. It then only accesses slots with matching fingerprints, skipping all the other slots. If there is no match, the key is definitely not present in the bucket.

4.3 Bucket Load Balancing

Segmentation reduces cache misses on the directory (by reducing its size). However, as we describe in Sections 2.3 and 6, this is at the cost of load factor: in a naive implementation the entire segment needs to be split if any bucket is full, yet other buckets in the segment may still have much free space. We observe that the key reason is load imbalance caused by the (inflexible) way buckets are selected for inserting new records, i.e., a key is only mapped to a single bucket. Dash uses a combination of techniques for new inserts to balance loads among buckets while limiting PM reads needed. Figure 3 shows how the insert operation works in Dash-EH at a high level, with three key techniques described below.

Balanced Insert. To insert a record whose key is hashed into bucket b ($hash(key) = b$), Dash probes both bucket b and $b+1$ and inserts the record into the bucket that is less full (Figure 3 step 1). The rationale behind is to improve load factor by amortizing the load of hot buckets while limiting PM accesses (at most two buckets).

Displacement. If both the target bucket b and probing bucket $b+1$ are full, Dash-EH tries to displace (move) a record from bucket b or $b+1$ to make room for the new record. With balanced insert, a record in bucket $n+1$ can be moved to $n+2$ if (1) it could be inserted to either bucket (i.e., $n+2$ is the probing bucket of the record being moved), *and* (2) bucket $n+2$ has a free slot. Thus, for a record with $hash(key) = b$ and both b and $b+1$ are full, we first try to find a record in $b+1$ whose $hash(key) = b+1$ and move it to bucket $b+2$. If such a record does not exist, we repeat for bucket b but move a record with $hash(key) = b-1$ (the target bucket). In essence, displacement follows a similar strategy to balanced insert, but is for existing records. We use a per-bucket `membership` bitmap (Figure 4) to indicate which records could be selected for displacement, accelerating this process.

Stashing. As shown in Figure 3, a tunable number of stash buckets follow the normal buckets in each segment. If a record cannot be inserted into its target bucket b nor the probing bucket $b+1$, we insert the record to a stash bucket; we call these records *overflow records*. Stash buckets use the same layout as that of normal buckets; probing of a stash bucket follows the same procedure as probing a normal bucket (see Section 4.2). While stashing can be effective in improving load factor, it could incur non-trivial overhead: the more stash buckets are used, the more CPU cycles and PM reads will be needed to probe them. This is especially undesirable for negative search and uniqueness check in insert operations, since both need to probe all stash buckets, despite it may be completely unnecessary.

To solve this problem, we try to set up record metadata including fingerprints in a normal bucket and only refer actual record access to the stash bucket. As Figure 4 shows, four additional fingerprints per bucket and overflow metadata (bits 208-240) are reserved for overflow records stored

in stash buckets. These metadata could indicate whether the searching key exists in the stash area, allowing early avoidance of access to stash buckets (and save PM bandwidth). We omit details here for space limitation. As Section 6 shows, using 2–4 stash buckets per segment can improve load factor to over 90% without imposing significant overhead.

4.4 Optimistic Concurrency

Dash employs optimistic locking, an optimistic flavor of bucket-level locking inspired by optimistic concurrency control [6]. Insert operations will follow traditional bucket-level locking to lock the affected buckets. Search operations are allowed to proceed without holding any locks (thus avoiding writes to PM) but need to verify the read record. For this to work, in Dash the lock consists of (1) a single bit that serves the role of "the lock" and (2) a version number for detecting conflicts (not to be confused with the version number in Figure 3 for instant recovery). The inserting thread will acquire bucket-level locks for the target and probing buckets by trying the `compare-and-swap` (`CAS`) instruction [5] to set the lock bit. After the insert is done, the thread releases the lock by (1) resetting the lock bit and (2) incrementing the version number by one, in one step using an atomic write.

To probe a bucket for a key, Dash first takes a snapshot of the lock word and checks whether the lock is being held by a concurrent writer (the lock bit is set). If so, it waits until the lock is released and repeats. Then it is allowed to read the bucket *without* holding any lock. Upon finishing its operations, the reader thread will read the lock word again to verify the version number did not change, and if so, it retries the entire operation as the record might not be valid as a concurrent write might have modified it.

4.5 Support for Variable-Length Keys

Dash stores pointers to variable-length keys, which is a common approach [10, 12, 22]. A knob is provided to switch between the inline (fixed-length keys up to 8 bytes) and pointer modes. Though dereferencing pointers may incur extra overhead, fingerprinting largely alleviates this problem. For negative search where the target key does not exist, no fingerprint will match and so key probing will not happen at all. For positive search, as we have discussed in Section 4.2, the amortized number of key load (therefore the number cache misses caused by following the key pointer) is one [12].

4.6 Record Operations

Now we present how Dash-EH performs insert, search and delete operations on PM with persistence guarantees.

Insert. Section 4.3 presented the high-level steps for insert; here we focus on the bucket-level. The inserting thread first writes and persists the new record in bucket, and then set up the metadata (fingerprint, allocation bitmap, counter and membership). The `CLWB` and fence are then issued to persist all the metadata. Once the corresponding bit in the bitmap is set, the record is visible to other threads. If a crash happens before the bitmap is persisted, the new record is regarded as invalid; otherwise, the record is successfully inserted. This allows us to avoid expensive logging while maintaining consistency.

Displacing a record needs two steps: (1) inserting it into the new bucket and (2) deleting it from the original bucket. In case a crash happens before step 2 finishes, a record will appear in both buckets. This necessitates a duplicate

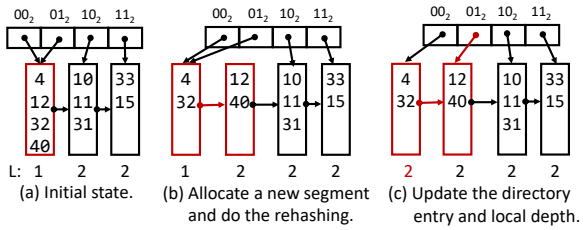


Figure 5: Segment split in Dash-EH; the global depth is 2.

detection mechanism upon recovery, which is amortized over runtime (see Section 4.8). If the insert has to happen in a stash bucket, we set the overflow metadata in the normal bucket. This cannot be done atomically with 8-byte writes and may need a (complex) protocol for crash consistency. We note that the overflow metadata is an optimization and does not influence correctness: records can still be found correctly even without it. So we do not explicitly persist it and rely on the lazy recovery mechanism to build it up gradually (described later).

Search. With balanced insert and displacement, a record could be inserted into its target bucket b where $b = hash(key)$ or its probing bucket $b + 1$. A search operation then has to check both if the record is not found in b. If neither bucket contains the record, it might be stored in a stash bucket. It will first probe the overflow metadata area of bucket b and access the stash buckets if necessary.

Delete. To delete a record in the bucket, we reset the corresponding bit in the allocation bitmap, decrement the counter and persist these changes. Then the slot becomes available for future reuse. To delete a record from a stash bucket, we also need to clear the corresponding overflow metadata stored in the target bucket.

4.7 Structural Modification Operations

After a thread exhausted all the options to insert a record into a bucket, it triggers a segment split that may expand the directory. To split a segment S, we (1) allocate a new segment N, (2) rehash keys in S and redistribute records in S and N, and (3) attach N to the directory and set the local depth of N and S. These steps cause the structure of the hash table to change and must be made crash consistent on PM while maintaining high performance.

For crash consistency, Dash-EH chains all segments using side links and each segment has a `state` variable that indicates whether the segment is in an SMO and whether it is the one being split or the new segment. An initial value of zero indicates the segment is not part of an SMO. Figure 5 shows an example. Note that Dash-EH uses the most significant bits (MSBs) of hash values to address and organize segments and buckets, which reduces cacheline flushes in the directory during splits [10]. To split a segment S, we first mark its `state` as `SPLITTING` and allocate a new segment N whose address is stored in the side link of S. N is then initialized to carry S's side link as its own. Its local depth is set to the local depth of S plus one. Then, we change N's `state` to `NEW` to indicate this new segment is part of a split SMO for recovery purposes (see Section 4.8). We rely on PM programming libraries (PMDK [4]) to atomically allocate *and* initialize the new segment; in case of a crash, the allocated PM block is guaranteed to be either owned by Dash or the allocator and will not be permanently leaked.

After initialization, we finish up step 2 by redistributing records between N and S. Records moved from S to N are deleted in S after they are inserted into N. Note that the rehashing/redistributing process does not need to be done atomically: if a crash happens in the middle of rehashing, upon (lazy) recovery we redo the rehashing process with uniqueness check to avoid repeating work for records that were already inserted into N before the crash; We describe more details later in Section 4.8. Figure 5(b) shows the state of the hash table after step 2. Then the directory entry for N and the local depth of S are updated as shown in Figure 5(c). Similarly, these updates are conducted using an atomic PMDK transaction which may use any approach such as lightweight logging. Many other systems avoid the use of logging to maintain high performance, largely because of the frequent pre-mature splits. But split is much rarer in Dash thanks to bucket load balancing that gives high load factor; this allows Dash-EH to employ PMDK transactions that abstracts away many details and eases implementation.

4.8 Instant Recovery

Dash provides truly instant recovery by requiring a constant amount of work (reading and possibly writing a one-byte counter), before the system is ready to accept user requests. We add a global version number V and a `clean` (boolean) marker shown in Figure 3, and a per-segment version number. The `clean` marker denotes whether the system was shutdown cleanly; V tells whether recovery (during runtime) is needed. Upon restart, if `clean` is false (no clean shut down), we increment V by one and start to handle requests. For both clean shutdown and crash cases, "recovery" only involves reading `clean` and possibly bumping V. The actual recovery work is amortized over segment accesses.

To access a segment, the accessing thread first checks whether the segment version matches V. If not, the thread (1) recovers the segment to a consistent state before doing its original operation (e.g., insert or search), and (2) sets the segment's version number to V so that future accesses can skip the recovery pass. Recovering a segment needs four steps: (1) clear bucket locks, (2) remove duplicate records caused by displacement, (3) rebuild overflow metadata, and (4) continue the ongoing SMO. With such lazy recovery approach, a segment is not recovered until it is accessed.

5. Dash FOR LINEAR HASHING

We present Dash-LH, Dash-enabled linear hashing that uses the building blocks discussed previously. We focus on the high-level design decisions specific to linear hashing; more details can be found in our original VLDB paper [9].

Figure 6 shows the overall structure of Dash-LH. Similar to Dash-EH, Dash-LH also uses segmentation and splits at the segment level. However, we follow the linear hashing approach to always split the segment pointed to by the `Next` pointer, which is advanced after the segment is split. Since the segment to be split is not necessarily full, full segments need to be able to temporarily accommodate overflow records, e.g., using linked lists. Traversing linked lists would incur many cache misses, which is a huge penalty for PM hash tables. We leverage stashing in Dash and use an adjustable number of stash buckets. In addition to a fixed number of stash buckets per segment, we store a linked list of stash buckets. A segment split is triggered whenever a stash bucket is allocated to accommodate overflow records. Dash-LH uses larger split unit (segment) and chaining unit (stash bucket rather than individual records), reducing chain length (therefore pointer chasing and cache misses). The overflow metadata and fingerprints further alleviate the performance penalty brought by the need to search stash buckets. To reduce directory size for better cache locality, we also introduce a hybrid expansion scheme which increases segment size exponentially. Overall, as we show in Section 6, Dash-LH can also achieve near-linear scalability on realistic workloads.

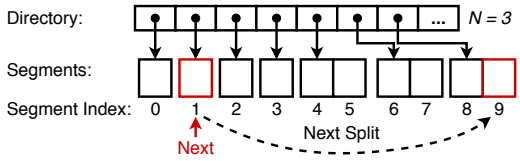


Figure 6: Overall design of Dash-enabled linear hashing.

6. EVALUATION

We evaluate Dash and compare it with state-of-the-art PM hash tables. Through experiments we confirm the following:

- Dash-enabled hash tables (Dash-EH and Dash-LH) scale well on multicore servers with real Optane DCPMM;
- The bucket load balancing techniques allow Dash to achieve high load factor while maintaining high performance;
- Dash recovers instantly with a small, constant amount of work upon restart, reducing service downtime.

Implementation. We implemented Dash-EH/LH using PMDK [4], which provides primitives for crash-safe PM management. The other hash tables under comparison (CCEH [10] and level hashing [22]) were both proposed based on DRAM emulation. We ported them to run on Optane DCPMM using their original code and PMDK; details are available in our original VLDB paper [9].

Setup. We run experiments on a server with a Intel Xeon Gold 6252 CPU clocked at 2.1GHz, 768GB of Optane DCPMM (6×128GB DIMMs on all six channels) in AppDirect mode, and 192GB of DRAM (6×32GB DIMMs). The CPU has 24 cores (48 hyperthreads) and 35.75MB of L3 cache. The server runs Arch Linux with kernel 5.5.3 and PMDK 1.7. All the code is compiled using GCC 9.2 with all optimization enabled. Threads are pinned to physical cores.

Parameters. For fair comparison, we set CCEH and level hashing to use the same parameters as in their original papers [10, 22]. Level hashing uses 128-byte (two cachelines) buckets. CCEH uses 16KB segments and 64-byte (one cacheline) buckets, with a probing length of four. Dash-EH and Dash-LH use 256-byte (four cachelines) buckets and 16KB segments. Each segment has two stash buckets.

Benchmarks. We stress test each hash table using microbenchmarks. Unless otherwise specified, for all runs we preload the hash table with 10 million records, then execute 190 million inserts (as an insert-only benchmark), 190 million positive search/negative search/delete operations back-to-back on the 200-million-record hash table. Similar to other work [10, 22], we use uniformly distributed random keys in our workloads. Due to space limitation, we omit the results over skewed workloads but all operations achieved similar and even better performance for higher cache hit ratios. For fixed-length key experiments, both keys and values are 8-byte

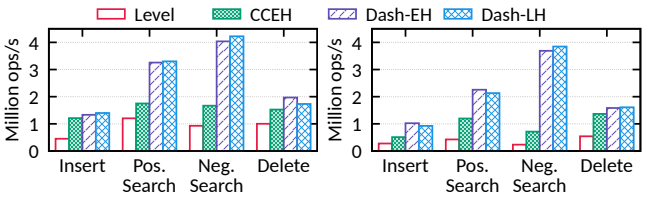


Figure 7: Single-thread performance under fixed-length keys (left) and variable-length keys (right).

integers; for variable-length key experiments, we use (pointers to) 16-byte keys and 8-byte values. The variable-length keys are pre-generated by the benchmark before testing.

6.1 Single-thread Performance

We begin with single-thread performance to understand the basic behaviors of each hash table. We first consider a read-only workload with fixed-length keys. Read-only results provide an upper bound performance on the hash tables since no modification is done to the data structure. They directly reflect the underlying design's cache efficiency and concurrency control overhead.

As Figure 7 shows, Dash-EH can outperform CCEH/level hashing by 1.9×/2.6× for positive search. Dash-LH and Dash-EH achieved similar performance because they use the same building blocks, with bounded PM accesses and lightweight concurrency control which reduces PM writes. For negative search, Dash achieved more significant improvement, benefitting from fingerprints and the overflow metadata.

For inserts, Dash and CCEH achieved similar performance (∼2.5× level hashing). Although CCEH has one fewer cacheline flush per insert than Dash, Dash's bucket load balancing strategy reduces segment splits, improving both performance and load factor. Level hashing exhibited much lower performance due to more PM reads. It also requires full-table rehashing that incurs many cacheline flushes. For deletes, Dash outperforms CCEH/level hashing by 1.2×/1.9× because of reduced cache misses.

The benefit of Dash is more prominent for variable-length keys. As Figure 7 shows, Dash-EH/LH are 2×/5× faster than CCEH/level hashing for positive search. The numbers are even greater (5×/15×) for negative search. These results again show the effectiveness of fingerprinting which all operations will benefit from, because they either directly query a key (search/delete) or require uniqueness check (insert).

6.2 Scalability

We test both individual operations and a mixed workload (20% insert and 80% search operations). For the mixed workload, we preload the hash table with 60 million records to allow search operations to access actual data.

Figure 8 plots how each hash table scales under a varying number of threads and fixed-length keys. For insert operations, level hashing exhibits the worst scalability mainly due to full-table rehashing, which is time-consuming on PM and blocks concurrent operations. With fingerprinting and bucket load balancing, Dash finishes uniqueness checks quickly and triggers fewer SMOs, with fewer PM accesses and interactions with the PM allocator. Although neither Dash-EH nor Dash-LH scales linearly as inserts inherently exhibit many random PM writes, Dash is the most scalable solution, being up to 1.3×/8.9× faster than CCEH/level hashing.

For search operations, Figures 8(b–c) show near-linear scalability for Dash-EH/LH. CCEH falls behind mainly due to its use of pessimistic locking which incurs large amount of PM writes even for read-only workloads (to acquire/release read locks). Level hashing uses a similar design but lock striping makes all the locks likely to fit into the CPU cache. Therefore, although level hashing has lower single-thread performance than CCEH, it still achieves similar performance to CCEH under multiple threads. Delete operations in Dash-EH, Dash-LH, CCEH and level hashing on 24 threads scale and improve over their single-threaded version by 8.4×, 9.8×, 6.1× and 14.7×, respectively. For the mixed workload on 24 threads, Dash outperforms CCEH/level hashing by 2.7×/9.0×.

We observed similar trends (but with widening gaps between Dash-EH/LH and CCEH/level hashing) for workloads using variable-length keys (not shown for limited space).

6.3 Load Factor

To compare different designs realistically, we observe how load factor changes after a sequence of inserts. We start with an empty hash table (load factor of 0) and measure the load factor after different numbers of records have been added to the hash tables. As shown in Figure 9, the load factor (y-axis) of CCEH fluctuates between 35% and 43%, because CCEH only conducts four cacheline probings before triggering a split. As we noted in Section 4.3, long probing lengths increase load factor at the cost of performance, yet short probing lengths lead to pre-mature splits. Compared to CCEH, Dash and level hashing can achieve high load factor because of their effective load factor improvement techniques. The "dipping" indicates segment splits/table rehashing is happening. We also observe that with two stash buckets, denoted as Dash-EH/LH (2), we achieve up to 80% load factor, while the number for using four stash buckets in Dash-EH (4) is 90%, matching that of level hashing.

6.4 Recovery

It is desirable for persistent hash tables to recover instantly after a crash or clean shutdown to reduce service downtime. We test recovery time by first loading a certain number of records and then killing the process and measuring the time needed for the system to be able to handle incoming requests. Table 1 shows the time needed for each hash table to get ready for handling incoming requests under different data sizes. The recovery time for Dash-EH/LH and level hashing are at sub-second level and does not scale as data size increases, effectively achieving instant recovery. For Dash-EH/LH the only needed work is to open the PM pool that is backing the hash table, and then read and possibly set the values of two variables. The recovery time for CCEH is linearly proportional to the data size because it needs to scan the entire directory upon recovery. As data size increases, so is the directory size, requiring more time on recovery.

6.5 Impact of PM Software Infrastructure

It has been shown that PM programming infrastructure can be a major overhead due to reasons such as page faults and cacheline flushes [8, 20]. We quantify its impact by running the same insert benchmark in Section 6.2 under two allocators (PMDK vs. a customized allocator) and two Linux kernel versions (5.2.11 vs. 5.5.3). Our customized allocator pre-allocates and pre-faults PM to remove page faults at runtime. An interesting finding is that Dash-LH

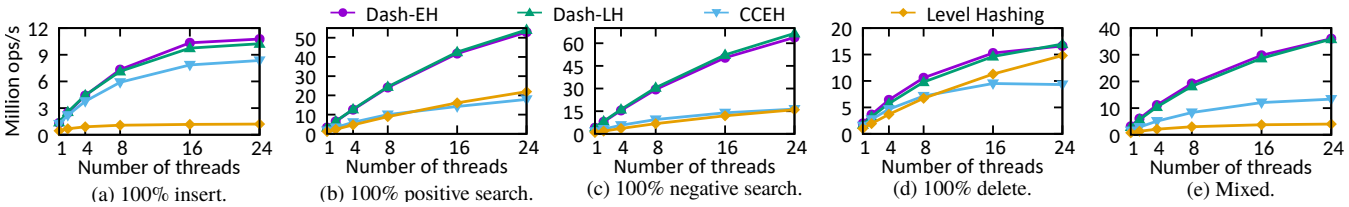


Figure 8: Throughput under different workloads with a varying number of threads and 8-byte keys and 8-byte values.

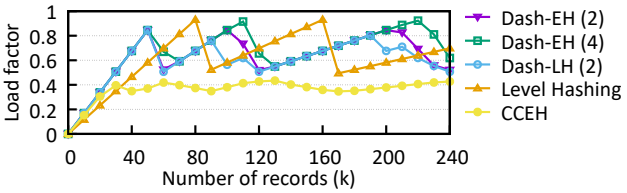


Figure 9: Load factor of different hashing schemes with respect to number of records inserted to the hash table.

Table 1: Recovery time (ms) vs. data size. CCEH's recovery time scales with data size whereas Dash and level hashing's remain constant.

Hash Table	Number of records (million)					
	40	80	160	320	640	1280
Dash-EH	57	57	57	57	57	57
Dash-LH	57	57	57	57	57	57
CCEH	113	165	262	463	870	1673
Level hashing	53	53	53	53	53	(53)

exhibited very low performance using PMDK allocator on Linux kernel 5.2.11 ($\sim$ 25% the number under 5.5.3). The reason was a bug in Linux kernel 5.2.11 that can cause large PM allocations to fall back to use 4KB pages, instead of 2MB huge pages (PMDK default).

Such results highlight the complexity of PM programming and call for careful design and testing in user and kernel spaces, given that the PM programming stack is evolving rapidly while practitioners and researchers have started to rely on them to build PM data structures.

7. CONCLUSION

Persistent memory brings new challenges to persistent hash tables in both performance (scalability) and functionality. We identify that the key is to reduce both unnecessary PM reads and writes, whereas prior work solely focused on reducing PM writes and ignored many practical issues such as PM management and concurrency control, and traded off instant recovery capability. Our solution is Dash, a holistic approach to scalable PM hashing. Dash combines both new and existing techniques, including (1) fingerprinting to reduce PM accesses, (2) optimistic locking, and (3) a novel bucket load balancing technique. Using Dash, we adapted extendible hashing and linear hashing to work on PM. On real Intel Optane DCPMM, Dash scales with up to $\sim$3.9$\times$ better performance than prior state-of-the-art, while maintaining desirable properties, including high load factor and sub-second level instant recovery.

8. REFERENCES

[1] R. Crooke and M. Durcan. A revolutionary breakthrough in memory technology. *3D XPoint Launch Keynote*, 2015.

[2] R. Fagin et al. Extendible hashing–a fast access method for dynamic files. *ACM TODS*, pages 315–344, 1979.

[3] S. Ghemawat and J. Dean. LevelDB. 2019. https://github.com/google/leveldb.

[4] Intel. Persistent Memory Development Kit. 2018. http://pmem.io/pmdk/libpmem/.

[5] Intel Corporation. Intel 64 and IA-32 architectures software developer's manual. 2015.

[6] H. T. Kung et al. On optimistic methods for concurrency control. *ACM TODS*, pages 213–226, 1981.

[7] P.-A. Larson. Dynamic hash tables. *CACM*, 1988.

[8] L. Lersch, X. Hao, I. Oukid, T. Wang, and T. Willhalm. Evaluating persistent memory range indexes. *PVLDB*, 13(4):574–587, 2019.

[9] B. Lu, X. Hao, T. Wang, and E. Lo. Dash: Scalable hashing on persistent memory. *PVLDB*, 13(8):1147–1161, 2020.

[10] M. Nam et al. Write-optimized dynamic hashing for persistent memory. *FAST*, pages 31–44, Feb. 2019.

[11] Oracle. MySQL. 2019. https://www.mysql.com/.

[12] I. Oukid et al. FPTree: A hybrid SCM-DRAM persistent and concurrent B-tree for storage class memory. *SIGMOD*, 2016.

[13] S. Pelley, T. F. Wenisch, B. T. Gold, and B. Bridge. Storage management in the NVRAM era. *PVLDB*, 7(2):121–132, 2013.

[14] PostgreSQL Global Development Group. PostgreSQL. 2019. http://www.postgresql.org/.

[15] Redis Labs. Redis. 2019. https://redis.io.

[16] S. Scargall. *Programming Persistent Memory: A Comprehensive Guide for Developers*. Apress, 2020.

[17] D. Schwalb, M. Dreseler, M. Uflacker, and H. Plattner. NVC-Hashmap: A persistent and concurrent hashmap for non-volatile memories. *IMDM*, pages 4:1–4:8, 2015.

[18] T. Wang and R. Johnson. Scalable logging through emerging non-volatile memory. *PVLDB*, 7(10), 2014.

[19] T. Wang, J. Levandoski, and P.-A. Larson. Easy lock-free indexing in non-volatile memory. *ICDE*, 2018.

[20] J. Yang et al. An empirical guide to the behavior and use of scalable persistent memory. *FAST*, 2020.

[21] H. Zhang et al. Reducing the storage overhead of main-memory oltp databases with hybrid indexes. *SIGMOD*, pages 1567–1581, 2016.

[22] P. Zuo, Y. Hua, and J. Wu. Write-optimized and high-performance hashing index scheme for persistent memory. *OSDI*, pages 461–476, Oct. 2018.