SIGMOD Officers, Committees, and Awardees

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

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

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

SIGMOD Executive Committee:

Divyakant Agrawal (Chair), Fatma Ozcan (Vice-chair), Rachel Pottinger (Treasurer), Juliana Freire (Previous SIGMOD Chair), K. Selçuk Candan (SIGMOD Conference Coordinator), Rada Chirkova (SIGMOD Record Editor), Chris Jermaine (ACM TODS Editor in Chief), Divesh Srivastava (2021 SIGMOD PC co-chair), Stratos Idreos (2021 SIGMOD PC co-chair), Leonid Libkin (Chair of PODS), Sihem Amer-Yahia (SIGMOD Diversity and Inclusion Coordinator), Curtis Dyreson (Information Director)

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 (SIGMOD Record), Georgia Koutrika (Blogging), Wim Martens (PODS), and Sourav S Bhowmick (SIGMOD Record)

SIGMOD Record Editor-in-Chief:

Rada Chirkova, NC State University

SIGMOD Record Associate Editors:

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

SIGMOD Conference Coordinator:

K. Selçuk Candan, Arizona State University

PODS Executive Committee:

Leonid Libkin (chair), Christoph Koch, Reinhard Pichler, Dan Suciu, Yufei Tao, Jan Van den Bussche

Sister Society Liaisons:

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

SIGMOD Awards Committee:

Sunita Sarawagi (Chair), Volker Markl, Renée Miller, H. V. Jagadish, Yanlei Diao, Stefano Ceri

Jim Gray Doctoral Dissertation Award Committee:

Vanessa Braganholo (co-chair), Viktor Leis (co-chair), Bailu Ding, Immanuel Trummer, Joy Arulraj, Jose Faleiro, Gustavo Alonso, Wolfgang Lehner

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** *Winne*r: Peter Bailis. *Honorable Mention*: Immanuel Trummer
- **2018** *Winne*r: Viktor Leis. *Honorable Mention*: Luis Galárraga and Yongjoo Park
- **2019** *Winne*r: Joy Arulraj. *Honorable Mention*: Bas Ketsman
- **2020** *Winner:* Jose Faleiro. *Honorable Mention:* Silu Huang
- **2021** *Winner:* Huanchen Zhang, *Honorable Mentions:* Erfan Zamanian, Maximilian Schleich, and Natacha Crooks

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

[Last updated: September 30, 2021]

Guest Editor's Notes

Welcome to the March 2022 issue of the ACM SIGMOD Record!

The new year of 2022 begins with a special issue on the **2021 ACM SIGMOD Research Highlights Awards**. These are awards for the database community to showcase a set of research projects that exemplify core database research. In particular, each of these projects addresses an important problem, represents a definitive milestone in solving the problem, and has the potential of significant impact. These awards also aim to make the selected works widely known in the database community, to our industry partners, and to the broader ACM community.

The awards committee and editorial board included Marcelo Arenas, Rada Chirkova, Wim Martens, Kenneth Ross, and Jun Yang. We solicited articles from PODS 2021, SIGMOD 2021, VLDB 2021, ICDE 2021, EDBT 2021, and ICDT 2021, as well as from community nominations. Through a careful review process ten articles were finally selected as 2021 Research Highlights. The authors of each article worked closely with the editorial board 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 2021 research highlights cover a broad set of topics, including (a) a significant step forward for query optimization that combines traditional wisdom with neural networks ("Bao: Making Learned Query Optimization Practical"); (b) a new approach that makes it easier for data processing systems to exploit high-speed networks ("DFI: The Data Flow Interface for High-Speed Networks"); (c) a foundation for future cloud information systems that aim at balancing performance and consistency ("FoundationDB: A Distributed Key Value Store"); (d) a new framework that learns deep contextualized representations on relational Web tables ("TURL: Table Understanding through Representation Learning"); (e) a highly scalable approach for embedding attributed networks on a single server ("No PANE, No Gain: Scaling Attributed Network Embedding in a Single Server"); (f) key innovations for bipartite matching that significantly improve the computation of the assignment costs ("Bipartite Matching: What to do in the Real World When Computing Assignment Costs Dominates Finding the Optimal Assignment"); (g) a system that combines the advantages of imperative and functional dataflow systems ("Imperative or Functional Control Flow Handling: Why not the Best of Both Worlds?"); (h) new insights for estimating quantiles in streams with strong theoretical guarantees, which are already available in the Apache Datasketches library ("Relative Error Streaming Quantiles"); (i) characterizations and answers to foundational questions on the consistency problem for relations under bag semantics ("Structure and Complexity of Bag Consistency"); and (j) an interesting new connection between counting the number of models of a CNF formula and the distinct elements in a data stream ("Model Counting Meets Distinct Elements in a Data Stream").

On behalf of the SIGMOD Record Editorial Board, I hope that you enjoy reading the March 2022 issue of the SIGMOD Record!

Wim Martens

March 2022

Making Learned Query Optimization Practical: A Technical Perspective

Volker Markl
Technische Universität Berlin, Germany
volker.markl@tu-berlin.de

Query optimization has been a challenging problem ever since the relational data model had been proposed. The role of the query optimizer in a database system is to compute an execution plan for a (relational) query expression comprised of physical operators whose implementations correspond to the operations of the (relational) algebra. There are many degrees of freedom for selecting a physical plan, in particular due to the laws of associativity, commutativity, and distributivity among the operators in the (relational) algebra, which necessitates our taking the order of operations into consideration. In addition, there are many alternative access paths to a dataset and a multitude of physical implementations for operations, such as relational joins (e.g., merge-join, nested-loop join, hash-join). Thus, when seeking to determine the best (or even a sufficiently good) execution plan there is a huge search space.

Query optimizers use a cost model to guide this search, which is usually a linear combination of the cardinality (i.e., the expected size of an intermediate result that needs to be computed when processing a query) and physical parameters, such as the I/O transfer cost and CPU processing cost. While the I/O transfer cost and CPU processing cost can easily be measured, it is very hard to accurately model the size of an intermediate result that may arise during query processing, in particular after many joins, selections, or projections with duplicate elimination have been applied to a data set.

Traditionally, query optimizers rely on statistics about a table and the database in conjunction with assumptions, such as the independence of predicates occurring in selections, the inclusion of one data set in another data set for key-foreign key joins, or uniformity when more informative statistics are unavailable. When these assumptions are invalid or the statistical information available is outdated or missing altogether, the cardinality estimates produced by the cost model of the query optimizer may be off, often by orders of magnitude. This in turn is the major cause for poor execution plans and suboptimal performance during query processing.

This problem has been well recognized by the research community, and in the early mid-90s researchers started to look into learning methods for specific statistical artefacts. In particular, to monitor cardinalities during query execution and exploit the learned information in future query compilations. Chen and Roussopulos [2] used query results to learn the statistical distributions of simple predicates after the execution of a query and accordingly adapt the coefficients via a curve-fitting function. Aboulnaga and Chaudhuri [1] introduced a query feedback loop, where the actual cardinalities monitored during execution are used to correct histograms.

LEO [4] pioneered the concept of a learning query optimizer, where a feedback loop is used to determine and adjust for discrepancies among cardinalities and other parameters of the cost model during all steps of the query execution plan. Recently, due to advances in machine learning and the emergence of Software 2.0, these ideas have received renewed interest. In particular, several works have proposed to use deep learning or reinforcement learning to compute the cardinalities or the entire cost model.

The Bao paper [3] is a consequent continuation of this line of research, which can take advantage of the prior knowledge hard-coded in the cost model of existing query optimizers, while leveraging the core ideas of learned optimizers (e.g., incorporating query feedback, learning from mistakes). BaoŠs key innovations include leveraging the existing query optimizer, using a deep convolutional neural network to predict the performance of queries and suggest hints, and leveraging Thompson sampling for plan selection. This approach allows Bao to be easily integrated and combined with existing systems, while being able to adjust to changes in the workload, data, and schema.

1. REFERENCES

[1] A. Aboulnaga and S. Chaudhuri. Self-tuning histograms: Building histograms without looking at data. In *SIGMOD*, 1999.
[2] C. M. Chen and N. Roussopoulos. Adaptive selectivity estimation using query feedback. In *SIGMOD*, 1994.
[3] R. Marcus, P. Negi, H. Mao, N. Tatbul, M. Alizadeh, and T. Kraska. Bao: Making learned query optimization practical. In *SIGMOD*, 2021.
[4] M. Stillger, G. M. Lohman, V. Markl, and M. Kandil. Leo - db2's learning optimizer. In *VLDB*, 2001.

Bao: Making Learned Query Optimization Practical

Ryan Marcus[12], Parimarjan Negi[1], Hongzi Mao[1],
Nesime Tatbul[12], Mohammad Alizadeh[1], Tim Kraska[1]
[1]MIT CSAIL [2]Intel Labs
{ryanmarcus, pnegi, hongzi, tatbul, alizadeh, kraska}@csail.mit.edu

ABSTRACT

Recent efforts applying machine learning techniques to query optimization have shown few practical gains due to substantive training overhead, inability to adapt to changes, and poor tail performance. Motivated by these difficulties, we introduce Bao (the Bandit optimizer). Bao takes advantage of the wisdom built into existing query optimizers by providing per-query optimization hints. Bao combines modern tree convolutional neural networks with Thompson sampling, a well-studied reinforcement learning algorithm. As a result, Bao automatically learns from its mistakes and adapts to changes in query workloads, data, and schema. Experimentally, we demonstrate that Bao can quickly learn strategies that improve end-to-end query execution performance, including tail latency, for several workloads containing long-running queries. In cloud environments, we show that Bao can offer both reduced costs and better performance compared with a commercial system.

1. INTRODUCTION

Query optimization is the task of transforming a user-issued declarative SQL query into an execution plan. Despite decades of study [30], query optimization remains a unsolved problem [17]. Several works have applied machine learning techniques to query optimization [32, 16, 19, 33, 35], often showing remarkable results. We argue that none of the techniques are yet practical, as they suffer from several fundamental problems:

1. **Long training time.** Most proposed machine learning techniques require an impractical amount of training data before they have a positive impact on query performance. For example, ML-powered cardinality estimators based on supervised learning require gathering precise cardinalities from the underlying data, a prohibitively expensive operation in practice (this is why we wish to estimate cardinalities in the first place). Reinforcement learning techniques must process thousands of queries before outperforming traditional optimizers, which can take on the order of days [19].

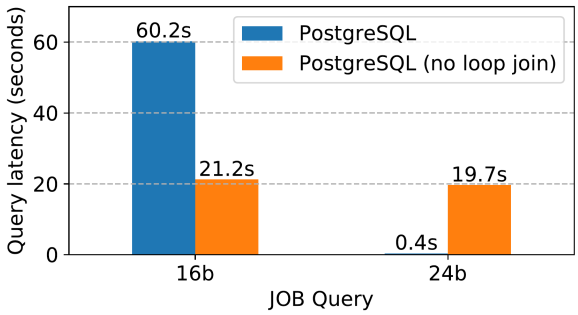


Figure 1: Disabling loop join in PostgreSQL can significantly improve (16b) or harm (24b) a particular query's performance. These example queries are from the Join Order Benchmark (JOB) [15].

2. **Handling change.** While performing expensive training operations once may already be impractical, changes in query workload, data, or schema can make matters worse. Supervised cardinality estimators must be retrained when data changes, or risk becoming stale. Several proposed reinforcement learning techniques require retraining when either the workload or the schema change [14, 20, 25, 19].

3. **Tail catastrophe.** Recent work has shown that learning techniques can outperform traditional optimizers *on average*, but often perform catastrophically (e.g., 100x regression in query performance) in the tail [19, 26, 24, 9]. This is especially true when training data is sparse. While some approaches offer statistical guarantees of their dominance in the average case [35], such failures, even if rare, are unacceptable in many real world applications.

4. **Black-box decisions.** While traditional cost-based optimizers are already complex, understanding query optimization is even harder when black-box deep learning approaches are used. Moreover, in contrast to traditional optimizers, current learned optimizers do not provide a way for database administrators to influence or understand the learned component's query planning.

5. **Integration cost.** To the best of our knowledge, all previous learned optimizers are still research prototypes, offering little to no integration with a real DBMS. None even supports all features of standard SQL, not to mention vendor specific features. Hence, fully integrating any learned optimizer into a commercial or open-source database system is not a trivial undertaking.

1.1 Bao

In [18], we introduced Bao (Bandit optimizer), the first learned optimizer which overcomes the aforementioned problems. This paper presents a short summary of Bao's key techniques and our experimental insights. Bao is fully integrated into PostgreSQL as an extension, and can be easily installed without the need to recompile PostgreSQL. The database administrator (DBA) just needs to download our open-source module, and even has the option to selectively turn the learned optimizer on or off for specific queries.

The core idea behind Bao is to avoid learning an optimizer from scratch. Instead, we take an existing optimizer (e.g., PostgreSQL's optimizer) and learn when to activate (or deactivate) some of its features on a query-by-query basis. In other words, Bao is a learned component that sits on top of an existing query optimizer in order to enhance query optimization, rather than replacing or discarding the traditional query optimizer altogether.

For instance, on a particular query, the PostgreSQL optimizer might under-estimate the cardinality for some joins and wrongly select a loop join when other join algorithms (e.g., merge join, hash join) would be more effective [15]. This occurs in query 16b of the Join Order Benchmark [15], and disabling loop-joins for this query yields a $3x$ performance improvement (see Figure 1). Yet, it would be wrong to always disable loop joins. For example, for query 24b, disabling loop joins causes the performance to degrade by almost $50x$, an arguably catastrophic regression.

At a high level, Bao tries to "correct" a traditional query optimizer by learning a mapping between an incoming query and the execution strategy the query optimizer should use for that query. We refer to these corrections – a subset of strategies to enable – as query hint sets. Effectively, through the provided hint sets, Bao limits and steers the search space of the traditional optimizer.

Our approach assumes a finite set of hint sets and treats each hint set as an arm in a contextual multi-armed bandit problem. Bao learns a model that predicts which hints will lead to good performance for a particular query. When a query arrives, our system selects a hint set, executes the resulting query plan, and observes a reward. Over time, Bao refines its model to more accurately predict which hint set will most benefit an incoming query. For example, for a highly selective query, Bao can automatically steer an optimizer towards a left-deep loop join plan (by restricting the optimizer from using hash or merge joins), and to disable loop joins for less selective queries.

By formulating the problem as a contextual multi-armed bandit, Bao can take advantage of Thompson sampling, a well-studied sample-efficient algorithm [3]. Because Bao uses an underlying query optimizer, Bao can potentially adapt to new data and schema changes just as well as the underlying optimizer. While other learned query optimization methods have to relearn what traditional query optimizers already know, Bao immediately starts learning to improve the underlying optimizer, and is able to reduce tail latency *even compared to traditional query optimizers.* In addition to addressing the practical issues of previous learned query optimization systems, Bao comes with a number of desirable features that were either lacking or hard to achieve in previous traditional and learned optimizers:

1. **Short training time.** In contrast to other deep-learning approaches, which can take days to train, Bao can outperform traditional query optimizers with much less training time ($\approx$ 1 hour). Bao achieves this by taking full advantage of existing query optimization knowledge, which was encoded by human experts into traditional optimizers available in DBMSes today. Moreover, Bao can be configured to start out using only the traditional optimizer and only perform training when the load of the system is low.

2. **Robustness to schema, data, and workload changes.** Bao can maintain performance even in the presence of workload, data, and schema changes. Bao does this by leveraging a traditional query optimizer's cost and cardinality estimates.

3. **Better tail latency.** While previous learned approaches either did not improve or did not evaluate tail performance, we show that Bao is capable of improving tail performance *by orders of magnitude* with as little as 30 minutes to a few hours of training.

4. **Interpretability and easier debugging.** Bao's decisions can be inspected using standard tools, and Bao can be enabled or disabled on a per-query basis. Thus, when a query misbehaves, an engineer can examine the query hint chosen by Bao and the decisions made by the underlying optimizer with `EXPLAIN`. If the underlying optimizer is functioning correctly, but Bao made a poor decision, Bao can be specifically disabled. Alternatively, Bao can be off by default, and only enabled on specific queries known to have poor performance with the underlying traditional query optimizer.

5. **Low integration cost.** Bao is easy to integrate into an existing database and often does not even require code changes, as most database systems already expose all necessary hints and hooks. Moreover, Bao builds on top of an existing optimizer and can thus support every SQL feature supported by the underlying database.

6. **Extensibility.** Bao can be extended by adding new query hints over time, without retraining. Additionally, Bao's feature representation can be easily augmented with additional information which can be taken into account during optimization, although this does require retraining. For example, when Bao's feature representation is augmented with information about the cache, Bao can learn how to change query plans based on the cache state. This is a desirable feature because reading data from cache is significantly faster than reading information off of disk, and it is possible that the best plan for a query changes based on what is cached. While integrating such a feature into a traditional cost-based optimizer may require significant engineering and hand-tuning, making Bao cache-aware is as simple as surfacing a description of the cache state.

Of course, Bao also has downsides. First, one of the most significant drawbacks is that query optimization time increases, as Bao must run the traditional query optimizer several times for each incoming query. A slight increase in optimization time is not an issue for problematic long-running queries, since the improved latency of the plan selected by Bao often greatly exceeds the additional optimization time. However, for very short running queries, increased optimization time can be an issue, especially if the application issues many such queries. Thus, Bao is ideally suited to workloads that are tail-dominated (e.g., 80% of query processing time is spent processing 20% of the queries) or

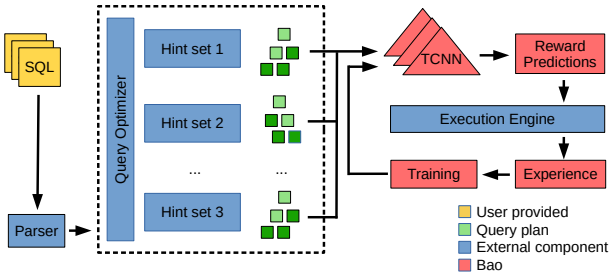


Figure 2: Bao system model

contain many long-running queries, although Bao's architecture also allows users to easily disable Bao for such short-running queries, or enable Bao exclusively for problematic longer-running queries. Second, by using only a limited set of hints, Bao has a restricted action space, and thus Bao is not always able to learn the best possible query plan. Despite this restriction, in our experiments, Bao is still able to significantly outperform traditional optimizers while training and adjusting to change orders-of-magnitudes faster than "unrestricted" learned query optimizers, like Neo [19].

In summary, the key contributions of this paper are:

- We introduce Bao, a learned system for query optimization that is capable of learning how to apply query hints on a case-by-case basis.
- For the first time, we demonstrate a learned query optimization system that outperforms both open source and commercial systems in cost and latency, all while adapting to changes in workload, data, and schema.

2. BAO ARCHITECTURE

On a high-level, Bao combines a tree convolution model [22], a neural network operator that can recognize important patterns in query plan trees [19], with Thompson sampling [3], a technique for solving contextual multi-armed bandit problems. This unique combination allows Bao to explore and exploit knowledge quickly. The architecture of Bao is shown in Figure 2.

Generating n query plans: When a user submits a query, Bao uses the underlying query optimizer to produce n query plans, one for each set of hint. Many DBMSes provide a wide range of such hints. While some hints can be applied to a single relation or predicate, Bao focuses only on query hints that are a boolean flag (e.g., disable loop join, force index usage). The sets of hints available to Bao must be specified upfront. Note that one set of hints could be empty, that is, using the original optimizer without any restriction.

Estimating the run-time for each query plan: Afterwards, each query plan is transformed into a vector tree (a tree where each node is a feature vector). These vector trees are fed into Bao's value model, a tree convolutional neural network [22], which predicts the quality (e.g., execution time) of each plan. To reduce optimization time, each of the n query plans can be generated and evaluated in parallel.

Selecting a query plan for execution: If we just wanted to execute the query plan with the best expected performance, we would train a model in a standard supervised fashion and pick the query plan with the best predicted performance. However, as our value model might be wrong, we might not always pick the optimal plan, and, as we never try alternative strategies, never learn when we are wrong. To balance the exploration of new plans with the exploitation of plans known to be fast, we use a technique called Thompson sampling [3] (see Section 3). It is also possible to configure Bao to explore a specific query offline and guarantee that only the best plan is selected during query processing (see [18]).

After a plan is selected by Bao, it is sent to a query execution engine. Once the query execution is complete, the combination of the selected query plan and the observed performance is added to Bao's experience. Periodically, this experience is used to retrain the predictive model, creating a feedback loop. As a result, Bao's predictive model improves, and Bao more reliable picks the best set of hints for each query. For workloads that cannot ever afford a query regression, this exploration can also be performed offline.

3. LEARNING FRAMEWORK

Here, we discuss Bao's learning approach. We first define Bao's optimization goal, and formalize it as a contextual multi-armed bandit problem. Then, we apply Thompson sampling, a classical technique used to solve such problems.

Bao models each hint set $HSet_i \in F$ in the family of hint sets F as if it were its own query optimizer: a function mapping a query $q \in Q$ to a query plan tree $t \in T$:

$$HSet_i : Q \to T$$

This function is realized by passing the query Q and the selected hint set $HSet_i$ to the underlying query optimizer. We refer to $HSet_i$ as this function for convenience. We assume that each query plan tree $t \in T$ is composed of an arbitrary number of operators drawn from a known finite set (i.e., that the trees may be arbitrarily large but all of the distinct operator *types* are known ahead of time).

Bao also assumes a user-defined performance metric P, which determines the quality of a query plan by executing it. For example, P may measure the execution time of a query plan, or may measure the number of disk operations performed by the plan.

For a query q, Bao must select a hint set to use. We call this selection function $B : Q \to F$. Bao's goal is to select the best query plan (in terms of the performance metric P) produced by a hint set. We formalize the goal as a regret minimization problem, where the regret for a query q, R_q, is defined as the difference between the performance of the plan produced with the hint set selected by Bao and the performance of the plan produced with the ideal hint set:

$$R_q = \Big(P(B(q)(q)) - \min_i P(HSet_i(q))\Big)^2 \qquad (1)$$

Contextual multi-armed bandits (CMABs) The regret minimization problem in Equation 1 is a contextual multi-armed bandit [39] problem. An agent must maximize their reward (i.e., minimize regret) by repeatedly selecting from a fixed number of *arms*. The agent first receives some contextual information (*context*), and must then select an arm. Each time an arm is selected, the agent receives a *payout*. The payout of each arm is assumed to be independent given the contextual information. After receiving the payout, the agent receives a new context and must select another arm. Each trial is considered independent.

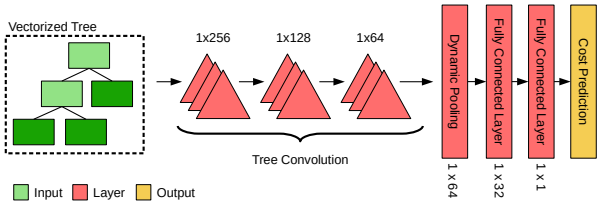


Figure 3: Bao prediction model architecture

For Bao, each "arm" is a hint set, and the "context" is the set of query plans produced by the underlying optimizer. Thus, our agent observes the query plans produced, chooses one of those plans, and receives a reward based on the resulting performance. Over time, our agent needs to improve its selection and get closer to choosing optimally (i.e., minimize regret). Doing so involves balancing exploration and exploitation: our agent must not always select a query plan randomly (as this would not help to improve performance), nor must our agent blindly use the first query plan it encounters with good performance (as this may leave significant improvements on the table). We use Thompson sampling [3], a well-studied technique for solving bandit problems.

3.1 Tree convolutional neural networks

We next explain Bao's predictive model, a tree convolution neural network [22, 19] responsible for estimating the quality of a query plan. Tree convolution is a composable and differentiable neural network operator specialized to work with tree-shaped inputs. Here, we give an intuitive overview of tree convolution, and refer to [19] for technical details and analysis of tree convolution on plan trees.

Human experts studying query plans learn to recognize good or bad plans by pattern matching: a pipeline of merge joins without any intermediate sorts may perform well, whereas a merge join on top of a hash join may induce a redundant sort or hash operation. Similarly, a hash join which builds a hash table over a very large relation may incur a spill, drastically slowing down execution. While none of this patterns are independently enough to decide if a query plan is good or bad, they do serve as useful indicators for further analysis; in other words, the presence or absence of such a pattern is a useful feature from a learning perspective. Tree convolution is precisely suited to recognize such patterns, and learns to do so *automatically, from the data itself.*

Tree convolution consists of sliding tree-shaped "filters" over a query plan tree (similar to image convolution, where filters are convolved with an image) to produce a transformed tree of the same size. These filters may look for patterns like pairs of hash joins, or an index scan over a small relation. Tree convolution operators are stacked in several layers. Later layers can learn to recognize more complex patterns, like a long chain of merge joins or a bushy tree of hash operators. Because of tree convolution's natural ability to represent and learn these patterns, we say that tree convolution represents a helpful *inductive bias* [21] for query optimization: that is, the structure of the network, not just its parameters, are tuned to the underlying problem.

The architecture of Bao's prediction model is shown in Figure 3. The query plan tree is passed through three layers of tree convolution. After the last layer of tree convolution, dynamic pooling [22] is used to flatten the tree structure into a single vector. Then, two fully connected layers are used to map the pooled vector to a performance prediction.

3.2 Training loop

Bao's training loop closely follows a classical Thompson sampling regime: when a query is received, Bao builds a query plan tree for each hint set and uses the current predictive model to select a plan to execute. After execution, that plan and the observed performance are added to Bao's experience. Periodically, Bao retrains its predictive model by sampling model parameters (i.e., neural network weights) to balance exploration and exploitation. Practical considerations specific to query optimization require a few deviations from the classical Thompson sampling regime, which we discuss next.

In classical Thompson sampling [34], the model parameters θ are resampled after every selection (query). In the context of query optimization, this is not practical for two reasons. First, sampling θ requires training a neural network, which is a time-consuming process. Second, if the size of the experience $|E|$ grows unbounded as queries are processed, the time to train the neural network will also grow unbounded, as the time required to perform a training epoch is linear in the number of training examples.

We use two techniques from prior work [4] to solve these issues. First, instead of resampling the model parameters (i.e., retraining the neural network) after every query, we only resample the parameters every nth query. This obviously decreases the training overhead by a factor of n by using the same model parameters for more than one query. Second, instead of allowing $|E|$ to grow unbounded, we only store the k most recent experiences in E. By tuning n and k, the user can control the tradeoff between model quality and training overhead to their needs.

We also introduce a new optimization, specifically useful for query optimization. On modern cloud platforms such as [1], GPUs can be attached and detached from a VM with per-second billing. Since training a neural network primarily uses the GPU, whereas query processing primarily uses the CPU, disk, and RAM, model training and query execution can be overlapped. When new model parameters need to be sampled, a GPU can be temporarily provisioned. Model training can then be offloaded to the GPU. Once model training is complete, the new model parameters can be swapped in for use when the next query arrives, and the GPU can be detached. Of course, users may also choose to use a machine with a dedicated GPU, or to offload model training to a different machine entirely, possibly with increased cost and network usage.

4. RELATED WORK

One of the earliest applications of learning to query optimization was Leo [32], which used successive runs of the similar queries to adjust histogram estimators. Recent approaches [16, 12, 27, 2] have learned cardinality estimations or query costs in a supervised fashion. Unsupervised approaches have also been proposed [38, 37]. While all of these works demonstrate improved cardinality estimation accuracy (potentially useful in its own right), they do not provide evidence that these improvements lead to better query plans. Ortiz et al. [26] showed that certain learned cardinality estimation techniques may improve mean performance on certain datasets, but tail latency is not evaluated. Negi et al. [24] showed how prioritizing training on cardinality estimations that have a large impact on query performance can improve estimation models.

	Size	Queries	WL	Data	Schema
IMDb	7.2 GB	5000	Dynamic	Static	Static
Stack	100 GB	5000	Dynamic	Dynamic	Static
Corp	1 TB	2000	Dynamic	Static[a]	Dynamic

[a] *The schema change did not introduce new data, but did normalize a large fact table.*

Table 1: Evaluation dataset sizes, query counts, and if the workload (WL), data, and schema are static or dynamic.

[20, 14] showed that, with sufficient training, reinforcement learning based approaches could find plans with lower costs (according to the PostgreSQL optimizer). [25] showed that the internal state learned by reinforcement learning algorithms are strongly related to cardinality. Neo [19] showed that deep reinforcement learning could be applied directly to query latency, and could learn optimization strategies that were competitive with commercial systems after 24 hours of training. However, none of these techniques are capable of handling changes in schema, data, or query workload, and none demonstrate improvement in *tail* performance. Works applying reinforcement learning to adaptive query processing [35, 11, 36] have shown interesting results, but are not applicable to existing, non-adaptive systems like PostgreSQL.

Our work is part of a recent trend in seeking to use machine learning to build easy to use, adaptive, and inventive systems, a trend more broadly known as machine programming [8]. In the context of data management systems, machine learning techniques have been applied to a wide variety of problems too numerous to list here, including index structures [13], data matching [7], workload forecasting [28], index selection [5], query latency prediction [6], and query embedding / representation [31, 10].

5. EXPERIMENTS

The key question we pose in our evaluation is whether or not Bao could have a positive, practical impact on real-world database workloads that include changes in queries, data, and/or schema. To answer this, we focus on quantifying not only query performance, but also on the dollar-cost of executing a workload (including the training overhead introduced by Bao) on cloud infrastructure against PostgreSQL and a commercial database system (Section 5.2). Here, we present only a short summary of our experimental evaluation; for a complete examination, see [18].

5.1 Setup

We evaluated Bao using the datasets listed in Table 1. The IMDb dataset is an augmentation of the Join Order Benchmark [15]. We created a new real-world datasets and workload called Stack, now publicly available.[1] – Stack contains over 18 million questions and answers from StackExchange websites (e.g., StackOverflow.com) over ten years. The Corp dataset is a dashboard workload executed over one month donated by an anonymous corporation. The Corp dataset contains 2000 unique queries issued by analysts. Half way through the month, the corporation normalized a large fact table, resulting in a significant schema change. We emulate this schema change by introducing the normalization after the execution of the 1000th query (queries after the 1000th expect the new normalized schema). The data remains static.

We use a "time series split" strategy for training and testing Bao. Bao is always evaluated on the next, never-before-seen query q_{t+1}. When Bao makes a decision for query q_{t+1}, Bao is only trained on data from earlier queries. Once Bao makes a decision for query q_{t+1}, the observed reward for that decision – and only that decision – is added to Bao's experience set. This strategy differs from previous evaluations in [19, 20, 14] because Bao is never allowed to learn from different decisions about the same query. In OLAP workloads where nearly-identical queries are frequently repeated (e.g., dashboards), this may be an overcautious procedure.

Unless noted, all experiments were performed on Google's Cloud Platform, using a N1-4 VM type and TESLA T4 GPU. Cost and time measurements include query optimization, model training (including GPU), and query execution. Costs are reported as billed by Google, and include startup times and minimum usage thresholds. Database statistics are fully rebuilt each time a new dataset is loaded.

We compare Bao against PostgreSQL and a commercial database system (ComSys) [29]. Both systems are configured and tuned according to their respective documentation and best practices guide; a consultant for ComSys double checked our configuration through small performance tests. For both baselines, we integrated Bao into the database using the original optimizer through hints. For example, we integrated Bao into ComSys by leveraging ComSys original optimizer with hints and executing all queries on ComSys.

Unless otherwise noted, queries are executed sequentially. We use 48 hint sets, which each use some subset of the join operators {hash join, merge join, loop join} and some subset of the scan operators {sequential, index, index only}. For a detailed description, see [18]. We found that setting the lookback window size to $k = 2000$ and retraining every $n = 100$ queries provided a good tradeoff between GPU time and query performance.

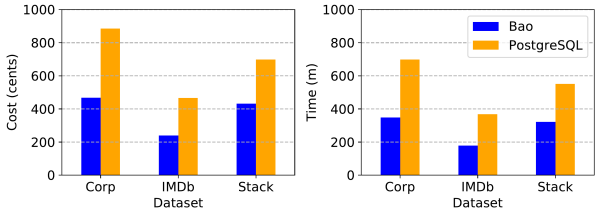
5.2 Is Bao practical?

In this first section, we evaluate if Bao is actually practical, and evaluate Bao's performance in a realistic warm-cache scenario; we augment each leaf node vector with caching information (see [18]).

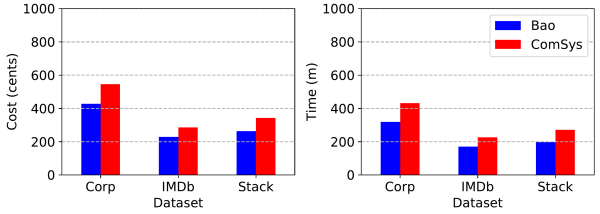
Cost and performance in the cloud Figure 4 shows the cost (left) and time required (right) to execute our three workloads in their entirety on the Google Cloud using an N1-16 VM. Bao outperforms PostgreSQL by almost 50% in cost and latency across the different datasets (Figure 4a). Note, that this *includes* the cost of training and the cost for attaching the GPU to the VM. Moreover, all datasets contain either workload, data, or schema changes, demonstrating Bao's adaptability to this common and important scenarios.

The performance improvement Bao makes on top of the commercial database is still significant though not as large (Figure 4b). Across the three dataset, we see an improvement of around 20%, indicating that ComSys optimizer is a much stronger baseline. Note, that the costs do *not* include the licensing fees for the commercial system. In our opinion, achieving a 20% cost and performance improvement over a highly capably query optimizer, which was developed over decades, without requiring any code changes to the database itself, is a very encouraging result.

[1] `https://rm.cab/stack`

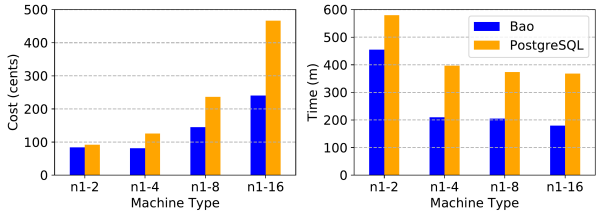


(a) Across our three evaluation datasets, Bao on the PostgreSQL engine vs. PostgreSQL optimizer on the PostgreSQL engine.

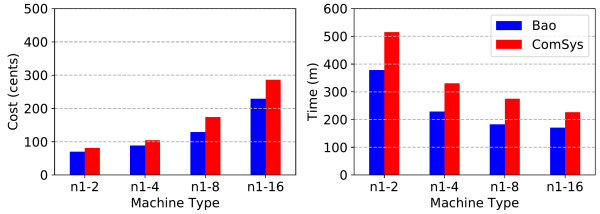


(b) Across our three evaluation datasets, Bao on the ComSys engine vs. ComSys optimizer on the ComSys engine.

Figure 4: Cost (left) and workload latency (right) for Bao and two traditional query optimizers across three different workloads on a N1-16 Google Cloud VM.



(a) Across four different VM types, Bao on the PostgreSQL engine vs. PostgreSQL optimizer on the PostgreSQL engine.



(b) Across four different VM types, Bao on the ComSys engine vs. ComSys optimizer on the ComSys engine.

Figure 5: Cost (left) and workload latency (right) for Bao and two traditional query optimizers across four different Google Cloud Platform VM sizes for the IMDb workload.

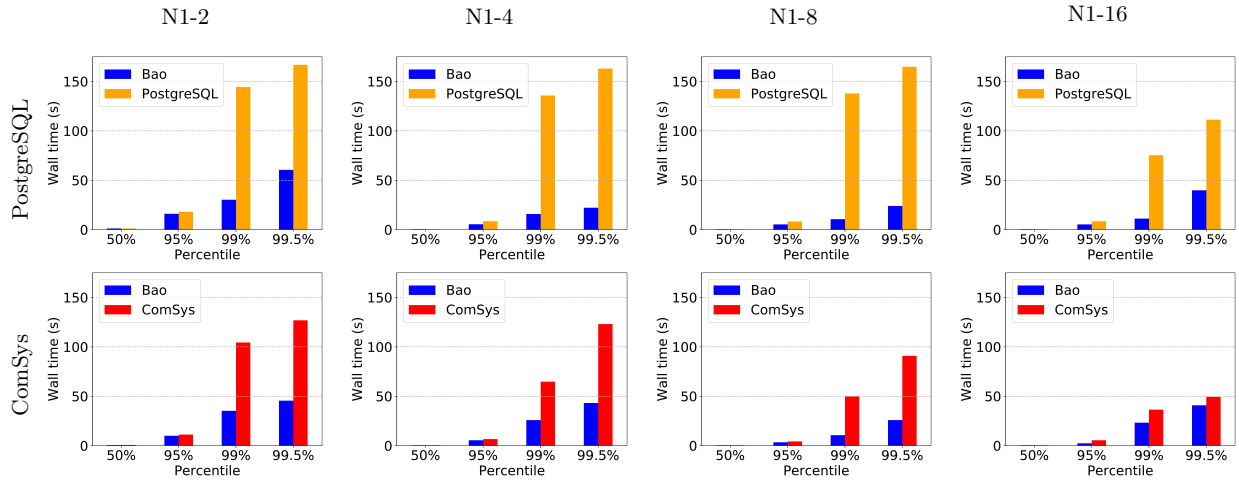
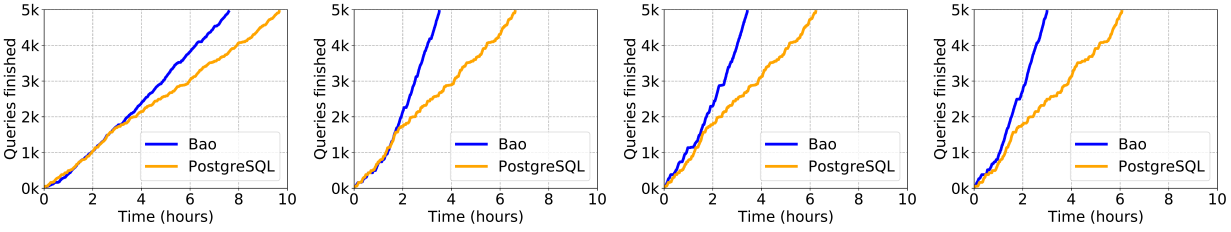


Figure 6: Percentile latency for queries, IMDb workload. Each column represents a VM type, from smallest to largest. The top row compares Bao against the PostgreSQL optimizer on the PostgreSQL engine. The bottom row compares Bao against a commercial database system on the commercial system's engine. Measured across the entire (dynamic) IMDb workload.



(a) VM type N1-2 (b) VM type N1-4 (c) VM type N1-8 (d) VM type N1-16

Figure 7: Number of IMDb queries processed over time for Bao and the PostgreSQL optimizer on the PostgreSQL engine. The IMDb workload contains 5000 unique queries which vary over time.

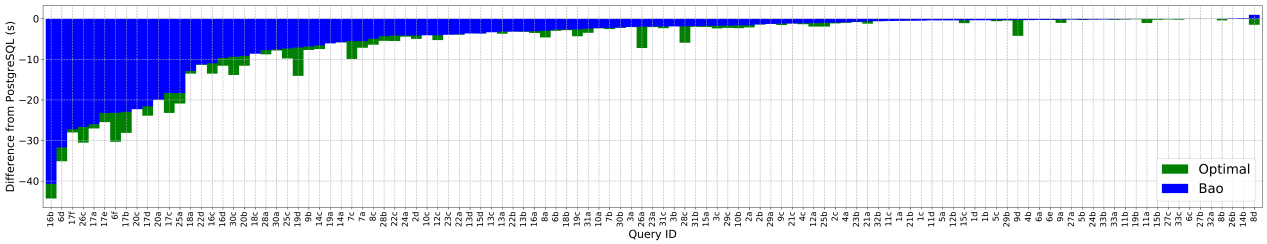


Figure 8: Absolute difference in query latency between Bao's selected plan and PostgreSQL's selected plan for the subset of the IMDb queries from the Join Order Benchmark [15] (lower is better).

Hardware type As a second experiment we varied the hardware type for the IMDb workload (Figure 5). For PostgreSQL (Figure 5a), the difference in both cost and performance is most significant with larger VM types (e.g., N1-16 vs. N1-8), suggesting the Bao is better capable of tuning itself towards the changing hardware than PostgreSQL. We *did* re-tune PostgreSQL for each hardware platform. Moreover, Bao itself benefits from larger machines as its parallelizes the execution of all the arms (discussed later).

Interestingly, whereas the benefits of Bao increase with larger machine sizes for PostgreSQL, it does not for the commercial system (Figure 5b). This suggests that the commercial system is more capable of adjusting to different hardware types, or perhaps that the commercial system is "by default" tuned for larger machine types. We note that the N1-2 machine type does not meet the ComSys vendor's recommended system requirements, although it does meet the vendor's minimum system requirements.

Tail latency analysis The previous two experiments demonstrate Bao's ability to reduce the cost and latency of an entire workload. Since practitioners are often interested in tail latency, here we examine the distribution of query latencies within the IMDb workload on multiple VM types. Figure 6 shows median, 95%, 99%, and 99.5% latencies for each VM type (column) for both PostgreSQL (top row) and the commercial system (bottom row). For each VM type, *Bao drastically decreases tail latencies when compared to the PostgreSQL optimizer.* For example, on an N1-8 instance, 99% latency fell from 130 seconds with the PostgreSQL optimizer to under 20 seconds with Bao. This suggests that most of the cost and performance gains from Bao come from reductions at the tail of the latency distribution. Compared with the commercial system, Bao always reduces tail latency, although the reduction is only significant on the smaller VM types where resources are more scarce.

It is important to note that Bao's improvement in tail latency (Figure 6) is primarily responsible for the overall improvements in workload performance (Figure 4). This is because these tail queries are disproportionately large contributors to workload latency (see Section 5.1 for a quantification). In fact, Bao hardly improves the median query performance at all ($< 5\%$). Thus, in a workload comprised entirely of such "median" queries, performance gains from Bao could be significantly lower. We next examine the worst case, in which the query optimizer is already making a near-optimal decision for each query. To demonstrate this, we executed the fastest 20% of queries from the IMDb workload using Bao and PostgreSQL. In this setup, Bao executed the restricted workload in 4.5m compared to PostgreSQL's 4.2m – this 18 second regression is attributable to additional overhead of Bao (quantified in [18]).

Training time and convergence A major concern with any application of reinforcement learning is convergence time. Figure 7 shows time vs. queries completed plots (performance curves) for each VM type while executing the IMDb workload. In all cases, Bao has similar performance to PostgreSQL for the first 2 hours, and exceeds the performance afterwards. Plots for the Stack and Corp datasets are similar. Plots comparing Bao against the commercial system are also similar, with slightly longer convergence times: 3 hours to exceed the performance of the commercial optimizer.

The IMDb workload is dynamic, yet Bao adapts to changes in the query workload. This is visible in Figure 7: Bao's performance curve remains straight after a short initial period, indicating that shifts in the query workload did not produce a significant change in query performance.

Query regression analysis Practitioners are often concerned with query regressions (e.g., when statistics change or a new version of an optimizer is installed) and thus naturally ask, would Bao cause regressions? Figure 8 shows the absolute performance improvement for Bao and what the theoretical optimal set of hints would be able to achieve (green) for each of the Join Order Benchmark (JOB) [15] queries, a subset of our IMDb workload. A negative value is a performance improvement, a positive value a regression. For this experiment, we trained Bao by executing the entire IMDb workload with the JOB queries removed, and then executed each JOB query without updating Bao's predictive model. That is, Bao has not seen any of the JOB queries before, and there was no predicate overlap. Of the 113 JOB queries, Bao only incurs regressions on three, and these regressions are all under 3 seconds. Ten queries see performance improvements of over 20 seconds. Of course, Bao (blue) does not always choose the optimal hint set (green).

6. CONCLUSION AND FUTURE WORK

This work introduced Bao, a bandit optimizer which steers a query optimizer using reinforcement learning. Bao is capable of matching the performance of commercial optimizers with as little as one hour of training time. We have demonstrated that Bao can reduce median and tail latencies, even in the presence of dynamic workloads, data, and schema.

In the future, we plan to investigate integrating Bao into cloud systems. With inspiring results from Microsoft [23], we believe that Bao can improve resource utilization in multi-tenant environments where disk, RAM, and CPU time are scarce resources. We additionally plan to investigate if Bao's

predictive model can be used as a cost model in a traditional database optimizer, enabling more traditional optimization techniques to take advantage of machine learning.

This research is supported by Google, Intel, and Microsoft as part of the MIT Data Systems and AI Lab (DSAIL) at MIT, and NSF IIS 1900933. This research was also sponsored by the United States Air Force Research Laboratory and the United States Air Force Artificial Intelligence Accelerator and was accomplished under Cooperative Agreement Number FA8750-19-2-1000. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the United States Air Force or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein.

7. REFERENCES

[1] Google Cloud Platform, https://cloud.google.com/.

[2] C. Anagnostopoulos and P. Triantafillou. Learning to accurately COUNT with query-driven predictive analytics. In *2015 IEEE International Conference on Big Data (Big Data)*, Big Data '15, pages 14–23, Oct. 2015.

[3] O. Chapelle and L. Li. An empirical evaluation of Thompson sampling. In *Advances in Neural Information Processing Systems*, NIPS'11, 2011.

[4] M. Collier and H. U. Llorens. Deep Contextual Multi-armed Bandits. *arXiv:1807.09809 [cs, stat]*, July 2018.

[5] B. Ding, S. Das, R. Marcus, W. Wu, S. Chaudhuri, and V. R. Narasayya. AI Meets AI: Leveraging Query Executions to Improve Index Recommendations. In *38th ACM Special Interest Group in Data Management*, SIGMOD '19, 2019.

[6] J. Duggan, O. Papaemmanouil, U. Cetintemel, and E. Upfal. Contender: A Resource Modeling Approach for Concurrent Query Performance Prediction. In *Proceedings of the 14th International Conference on Extending Database Technology*, EDBT '14, pages 109–120, 2014.

[7] R. C. Fernandez and S. Madden. Termite: A System for Tunneling Through Heterogeneous Data. In *AIDM @ SIGMOD 2019*, aiDM '19, 2019.

[8] J. Gottschlich, A. Solar-Lezama, N. Tatbul, M. Carbin, M. Rinard, R. Barzilay, S. Amarasinghe, J. B. Tenenbaum, and T. Mattson. The three pillars of machine programming. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2018, pages 69–80, Philadelphia, PA, USA, June 2018. Association for Computing Machinery.

[9] R. B. Guo and K. Daudjee. Research challenges in deep reinforcement learning-based join query optimization. In *Proceedings of the Third International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, aiDM '20, pages 1–6, Portland, Oregon, June 2020. Association for Computing Machinery.

[10] S. Jain, B. Howe, J. Yan, and T. Cruanes. Query2Vec: An Evaluation of NLP Techniques for Generalized Workload Analytics. *arXiv:1801.05613 [cs]*, Feb. 2018.

[11] T. Kaftan, M. Balazinska, A. Cheung, and J. Gehrke. Cuttlefish: A Lightweight Primitive for Adaptive Query Processing. *arXiv preprint*, Feb. 2018.

[12] A. Kipf, T. Kipf, B. Radke, V. Leis, P. Boncz, and A. Kemper. Learned Cardinalities: Estimating Correlated Joins with Deep Learning. In *9th Biennial Conference on Innovative Data Systems Research*, CIDR '19, 2019.

[13] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis. The Case for Learned Index Structures. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, New York, NY, USA, 2018. ACM.

[14] S. Krishnan, Z. Yang, K. Goldberg, J. Hellerstein, and I. Stoica. Learning to Optimize Join Queries With Deep Reinforcement Learning. *arXiv:1808.03196 [cs]*, Aug. 2018.

[15] V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann. How Good Are Query Optimizers, Really? *PVLDB*, 9(3):204–215, 2015.

[16] H. Liu, M. Xu, Z. Yu, V. Corvinelli, and C. Zuzarte. Cardinality Estimation Using Neural Networks. In *Proceedings of the 25th Annual International Conference on Computer Science and Software Engineering*, CASCON '15, pages 53–59, Riverton, NJ, USA, 2015. IBM Corp.

[17] G. Lohman. Is Query Optimization a '"Solved" Problem? In *ACM SIGMOD Blog*, ACM Blog '14, 2014.

[18] R. Marcus, P. Negi, H. Mao, N. Tatbul, M. Alizadeh, and T. Kraska. Bao: Making Learned Query Optimization Practical. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, China, June 2021.

[19] R. Marcus, P. Negi, H. Mao, C. Zhang, M. Alizadeh, T. Kraska, O. Papaemmanouil, and N. Tatbul. Neo: A Learned Query Optimizer. *PVLDB*, 12(11):1705–1718, 2019.

[20] R. Marcus and O. Papaemmanouil. Deep Reinforcement Learning for Join Order Enumeration. In *First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, aiDM @ SIGMOD '18, Houston, TX, 2018.

[21] T. M. Mitchell. The Need for Biases in Learning Generalizations. Technical report, 1980.

[22] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin. Convolutional Neural Networks over Tree Structures for Programming Language Processing. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI '16, pages 1287–1293, Phoenix, Arizona, 2016. AAAI Press.

[23] P. Negi, M. Interlandi, R. Marcus, M. Alizadeh, T. Kraska, M. Friedman, and A. Jindal. Steering Query Optimizers: A Practical Take on Big Data Workloads. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, pages 2557–2569, Virtual Event China, June 2021. ACM.

[24] P. Negi, R. Marcus, H. Mao, N. Tatbul, T. Kraska, and M. Alizadeh. Cost-Guided Cardinality Estimation: Focus Where it Matters. In *Workshop on Self-Managing Databases*, SMDB @ ICDE '20, 2020.

[25] J. Ortiz, M. Balazinska, J. Gehrke, and S. S. Keerthi. Learning State Representations for Query Optimization with Deep Reinforcement Learning. In *2nd Workshop on Data Managmeent for End-to-End Machine Learning*, DEEM '18, 2018.

[26] J. Ortiz, M. Balazinska, J. Gehrke, and S. S. Keerthi. An Empirical Analysis of Deep Learning for Cardinality Estimation. *arXiv:1905.06425 [cs]*, Sept. 2019.

[27] Y. Park, S. Zhong, and B. Mozafari. QuickSel: Quick Selectivity Learning with Mixture Models. *arXiv:1812.10568 [cs]*, Dec. 2018.

[28] A. Pavlo, E. P. C. Jones, and S. Zdonik. On Predictive Modeling for Optimizing Transaction Execution in Parallel OLTP Systems. *PVLDB*, 5(2):86–96, 2011.

[29] A. G. Read. DeWitt clauses: Can we protect purchasers without hurting Microsoft. *Rev. Litig.*, 25:387, 2006.

[30] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access Path Selection in a Relational Database Management System. In J. Mylopolous and M. Brodie, editors, *SIGMOD '79*, SIGMOD '79, pages 511–522, San Francisco (CA), 1979. Morgan Kaufmann.

[31] Shrainik Jain, Jiaqi Yan, Thiery Cruanes, and Bill Howe. Database-Agnostic Workload Management. In *9th Biennial Conference on Innovative Data Systems Research*, CIDR '19, 2019.

[32] M. Stillger, G. M. Lohman, V. Markl, and M. Kandil. LEO - DB2's LEarning Optimizer. In *VLDB*, VLDB '01, pages 19–28, 2001.

[33] J. Sun and G. Li. An end-to-end learning-based cost estimator. *Proceedings of the VLDB Endowment*, 13(3):307–319, Nov. 2019.

[34] W. R. Thompson. On the Likelihood that One Unknown Probability Exceeds Another in View of the Evidence of Two Samples. *Biometrika*, 1933.

[35] I. Trummer, S. Moseley, D. Maram, S. Jo, and J. Antonakakis. SkinnerDB: Regret-bounded Query Evaluation via Reinforcement Learning. *PVLDB*, 11(12):2074–2077, 2018.

[36] K. Tzoumas, T. Sellis, and C. Jensen. A Reinforcement Learning Approach for Adaptive Query Processing. *Technical Reports*, June 2008.

[37] Z. Yang, A. Kamsetty, S. Luan, E. Liang, Y. Duan, X. Chen, and I. Stoica. NeuroCard: One Cardinality Estimator for All Tables. *arXiv:2006.08109 [cs]*, June 2020.

[38] Z. Yang, E. Liang, A. Kamsetty, C. Wu, Y. Duan, X. Chen, P. Abbeel, J. M. Hellerstein, S. Krishnan, and I. Stoica. Deep unsupervised cardinality estimation. *Proceedings of the VLDB Endowment*, 13(3):279–292, Nov. 2019.

[39] L. Zhou. A Survey on Contextual Multi-armed Bandits. *arXiv:1508.03326 [cs]*, Feb. 2016.

Technical perspective: DFI: The Data Flow Interface for High-Speed Networks

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

Optimizing data movement has always been one of the key ways to get a data processing system to perform efficiently. Appearing under different disguises as computers evolved over the years, the issue is today as relevant as ever. With the advent of the cloud, data movement has become *the* bottleneck to address in any data processing system. In the cloud, compute and storage are typically disaggregated, with a network in between. In addition, cloud systems are scale-out, i.e., performance is obtained by parallelizing across machines, which also involves network communication. And while it is possible to use machines with large amounts of memory, the pricing models and the virtualized nature of the cloud tends to favor clusters of smaller computing nodes. Nowadays, the problem of optimizing data movement has become the problem of using the network as efficiently as possible.

Unfortunately, optimizing the network is easier said than done. Network communication is actually computationally quite expensive. As network bandwidth grows and the potential number of concurrent flows (connections) increases, the computational power required to run the network protocol can be quite high. In the cloud, this problem is made more acute by the need to support many concurrent virtual machines on the same physical node (implying many more connections) and the necessity of virtualizing the network, which adds considerable overhead. Such an overhead has led to the proliferation of smart NICs where most, if not all, of the computation related to the network protocol is offloaded, thereby freeing the CPU from the task.

It is in this context that Remote Direct Memory Access (RDMA) plays a big role. As the name implies, RDMA enables the transfer of data directly across the memories of the sender and receiver with no (single-sided operations) or only minimal (two-sided operations) intervention of the CPU since the transfer is managed by the NIC. Doing so removes many of the inefficiencies of conventional network protocols (involvement of the operating system which introduces overhead for system calls and context switches, need to copy the data from the network buffers to the operating system and then to user space, need to reserve CPU capacity to process network packages, etc.), leading to significant lower latencies.

Unfortunately, nothing is perfect. RDMA offers only a low level interface that requires the developer to manage a great deal of complexity. In high performance computing, this has been addressed through the development of libraries such as MPI. However, MPI is surprisingly unsuitable for conventional applications for a wide variety of reasons (the very restrictive computational model, poor support for multithreading, etc.). Thus, it seems to make sense to develop an equivalent to MPI for data processing using RDMA. The paper highlighted in the next section does exactly this by proposing the DFI (*Data Flow Interface*), a library that uses RDMA in the background to provide a declarative way to specify data exchanges across nodes specifically tailored to data processing applications. DFI abstracts away many of the elements of RDMA and replaces them with data structures and interfaces more suitable to data management. For instance, RDMA requires to register memory in advance with the NIC so that the NIC can actually operate on that region of memory without CPU or OS involvement. This adds overhead to every exchange and is difficult to manage. In data processing, the amount of data to transfer depends on the selectivity, which varies greatly from query to query. This would imply that a large part of memory would have to be reserved for RDMA exchanges in case a lot of data is involved even if it is not often used. Given the demand for main memory in today's systems, such overprovisioning would be highly problematic. The DFI addresses this problem by automatically introducing and managing ring buffers where data can be continuously be sent without having to register new regions mid-way through the exchange or having to resort to overprovisioning. Similarly, using RDMA directly, developers need to take care of partitioning the data across nodes and of routing the data as needed. DFI takes care of such tasks by supporting pluggable partition operators and also letting the developer indicate the type of exchange to implement in a declarative manner.

The paper makes a convincing case that DFI is more suitable for data processing than existing solutions by presenting several use cases where DFI takes care of complex tasks that RDMA would force the developer to manage manually. The performance numbers are impressive, showing the DFI does not add any significant overhead and that sometimes leads to better results as some of the complex steps required when using RDMA directly are not longer needed. With the growing number of systems and research relying on RDMA, DFI is a valuable contribution that opens up many possibilities for building systems but also by enabling research on network optimizations for data processing.

DFI: The Data Flow Interface for High-Speed Networks

Lasse Thostrup
TU Darmstadt

Jan Skrzypczak*
Zuse Institute, Berlin

Matthias Jasny
TU Darmstadt

Tobias Ziegler
TU Darmstadt

Carsten Binnig
TU Darmstadt

ABSTRACT

In this paper, we propose the Data Flow Interface (DFI) as a way to make it easier for data processing systems to exploit high-speed networks without the need to deal with the complexity of RDMA. By lifting the level of abstraction, DFI factors out much of the complexity of network communication and makes it easier for developers to declaratively express how data should be efficiently routed to accomplish a given distributed data processing task. As we show in our experiments, DFI is able to support a wide variety of data-centric applications with high performance at a low complexity for the applications.

1 Introduction

Motivation: Scale-out data processing systems are the typical architecture used today by many systems to process large data volumes since they allow applications to increase compute and memory capacities by simply adding further processing nodes. However, a typical bottleneck in scale-out systems is the network which often slows down the speed of data processing if communication is in the critical path. For distributed in-memory systems this might lead to degraded performance when adding more nodes.

However, this changed with the advent of high-speed networks such as InfiniBand. Network bandwidth increased almost up to the speed of main memory and latencies dropped by orders of magnitude [4], making scale-out solutions more competitive. However, blindly upgrading to faster networks does often not directly translate into performance gains, as there is a plenitude of aspects to consider to achieve a good performance for distributed data processing systems.

One particular important aspect to efficiently use high-speed networks is to redesign data processing systems to leverage remote direct memory access (RDMA) as a low overhead communication protocol. RDMA provides kernel bypass and zero-copy making data transfers less expensive than classical network stacks such as TCP/IP [9]. In recent years, industry and academia have thus started to adapt scale-out data processing systems in order to make use of RDMA. As a result, significant speed-ups have been shown for a wide range of data processing systems ranging from key-value stores [11, 15, 17], over distributed DBMSs (for OLTP and OLAP) [4, 12, 23] to Big Data systems and Distributed Machine Learning [22].

However, using RDMA is complicated because it provides only low-level abstractions (called RDMA verbs) for data processing. Hence, redesigning data processing systems for RDMA often requires significant efforts to take care of many low-level detail choices [4, 3, 12, 25] regarding remote memory and connection management as well as other decisions such as which RDMA verbs to use for which type of workload.

Contribution: In this paper, we propose the Data Flow Interface (DFI) as a way to make it easier for data processing systems to exploit high-speed networks. Accordingly, DFI defines abstractions and interfaces suited to a broad class of data-intensive applications, yet simple enough for practical implementation with predictable performance and low overhead relative to "hand-tuned", ad hoc alternatives. In designing a high-level interface tailored to data processing, we adopt the approach taken by the high-performance community for MPI [10] to provide a simple yet effective interface for high-speed networks. However, since MPI has been designed for computation-intensive workloads such as large-scale simulations, it comes with many design choices that are not optimal for data-intensive workloads [13]. Consequently, MPI has seen only very limited adoption for data processing systems [2].

In brief, the main idea of DFI is that data movements are represented as *flows*. DFI flows are an abstraction providing primitives for efficient network communication. These primitives are intended to be used as a foundation for building data-intensive systems and provide many benefits over MPI (e.g., thread-centricity and pipelined communication). By lifting the level of abstraction, DFI flows not only hide much of the low-level complexity of network communication but also allow developers to declaratively express how data should be efficiently routed to accomplish a given distributed data processing task. Moreover, DFI flows allow developers to specify *optimization hints*; e.g., to maximize bandwidth-utilization or minimize network latency of transfers. By using flows as the main abstraction, DFI supports a wide variety of data-centric applications ranging from bandwidth-sensitive distributed OLAP to more latency-sensitive work-

*Work done while at Zuse Institute, Berlin

loads such as distributed OLTP or replication with consensus protocols.

Recently, the need for better interfaces to high-speed networks has also been discussed in a vision paper [1]. We, however, are the first paper that provide a concrete suggestion and a full implementation for an interface that can enable a broad class of data-centric applications to make efficient use of modern networks. Moreover, there have also been several other attempts to build libraries for data processing over high-speed networks [7, 8, 5]. For example, FaRM [7] and GAM [5] provide a programming model based on a shared address space which focuses on supporting latency-sensitive workloads (e.g., such as distributed transactions). Another example is L5 [8], which target the communication between clients and servers to replace traditional client-centric communication libraries such as ODBC. Different from those libraries, as mentioned before, DFI flows aim to be a much more general abstraction that can support a broader class of data-centric applications.

In summary, this paper makes the following contributions:

- First, we present the design of DFI based on the general abstraction of flows that allow developers to declaratively specify the communication behavior of distributed systems by defining its topology (1:1, N:1, 1:N and N:M) as well as providing other properties for execution.
- Second, we provide a first implementation of DFI[1] for an InfiniBand-based networking stack and discuss how the high-level abstractions of DFI are being mapped to the low-level implementations using RDMA.
- Third, we provide an evaluation of our DFI implementation and demonstrate that DFI is able to efficiently utilize the network but also showcase that DFI can provide high performance for different data-centric applications.

Outline: The remainder of this paper is structured as follows: In Section 2, we first give an overview of two existing interfaces, RDMA verbs and MPI. Afterwards, in Section 3 we present an overview of DFI before we discuss details of the programming model in Section 4 as well as our implementation for InfiniBand in Section 5. Finally, we conclude with our evaluation in Section 6, details on how to integrate DFI in Section 7 and a summary in Section 8.

2 Existing Interfaces

In this section we give a short overview of existing interfaces namely the standard RDMA verbs interface native to the InfiniBand network stack and the Message Passing Interface (MPI), the de facto standard in the HPC community.

2.1 RDMA Verbs

The InfiniBand RDMA verb interface is a low-level interface providing low latency and high bandwidth communication. The interface exposes one-sided verbs (*write*, *read* & *atomics*) and two-sided verbs (*send* & *receive*) which refer to the involvement of end-points (i.e., one-sided verbs only involves the CPU of the sender). The high performance of RDMA is in general achieved by the asynchronous nature of RDMA, making it possible to pipeline computation and communication such that the CPU is not busy idling during network communication. To issue RDMA verbs (one- or two-sided), the application has to register a memory region in which the RNIC can directly access memory, leaving communication related memory-management a responsibility of the application. Moreover, due to the RDMA verb interface's very low abstraction level it provides also a huge design space. This requires applications need to carefully explore this design space and to optimally make use the available low-level options [8, 25, 11, 24].

[1] `https://github.com/DataManagementLab/DFI-public`

2.2 Message Passing Interface

The Message Passing Interface (MPI) is widely used by the HPC community as a high-level abstraction for high-speed networks, and has through many years of development reached a mature and industrial-strength quality. It has however seen limited adoption in data management systems due to its synchronization heavy block-synchronous-parallel processing model and poor multi-threading performance [20].

3 DFI Overview

In this section, we first highlight the central design goals of DFI before we discuss the flow-based programming model, as well as the high-level idea of the execution model behind flows.

3.1 Key Design Principles

The aim of DFI is to provide a high-level abstraction that provides efficient support for a broad set of data processing systems. In the following, we present the key design principles of DFI to ideally support the needs of these systems:

(1) Pipelining: Different from MPI, which targets compute-centric applications such as distributed simulations, many data-centric applications are often dominated by data transfers (i.e., data shuffling). For this reason, it is shown to be crucial that computation and communication can be overlapped [3].

(2) Thread-centricity: Multi-threading is essential not only in achieving high degrees of parallelism in modern data-centric architectures but also to saturate the network as mentioned before. Hence, different from MPI, DFI should be designed from ground up to enable a thread-centric execution and communication model.

(3) Low-overhead synchronization: Another important aspect that goes along with thread-centricity is that DFI aims to provide low-overhead synchronization between sender and receiver threads as well as between sender threads that target the same receiver. By providing low-overhead synchronization, DFI thus should enable scalability to a high number of sender and receiver threads.

(4) Declarative optimization: A last important goal is that DFI exposes parameters as a handle for applications to declare what optimizations are desired. Examples of such optimizations are whether applications are bandwidth or latency sensitive, but also other guarantees such as global ordering of messages when data is send across flows (which is important, for example, for data replication protocols).

3.2 Flow-based Programming Model

At the center of the abstraction are DFI's flows. Flows encapsulate the movement of data between end-points in a

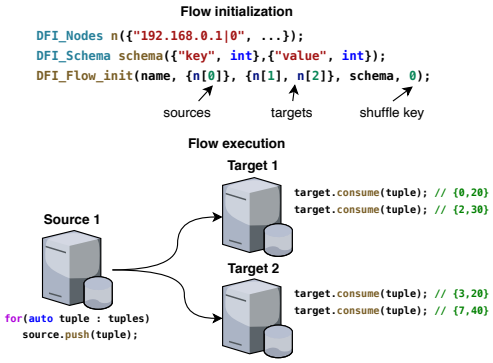


Figure 1: DFI's Programming and Execution Model. Example of flow initialization for setting up a shuffle based flow. The flow execution exemplifies the tuple-based push and consume primitives on DFI.

distributed application, by exposing *sources* and *targets* as data entry and exit points on a per thread-level. This simple abstraction allows applications to compose potentially complex communication topologies, including both point-to-point, one-to-many, many-to-one and many-to-many communications between worker threads of multiple nodes. As we show later in this section, the flow abstraction is powerful enough to support a wide range of data processing use-cases such as distributed join algorithms, but also consensus protocols.

In the following, we provide an example of a concrete many-to-many flow type in DFI, which is one out of multiple other flow types as we discuss later. The most common many-to-many communication in data processing systems is arguably key-based shuffling of data across multiple sources and targets. An example of such a shuffle flow in DFI is illustrated in Figure 1.

As we see in the example, before a flow can be used it first has to be initialized by specifying a unique flow name identifier, location of source and target threads identified with the node address and a thread ID in DFI_Nodes, the schema of tuples that are transferred and on which key the tuples should be shuffled (see upper part of Figure 1). Note, it is also possible for applications to specify application-specific partition functions, but as default a simple key-based hash function is used to partition the tuples across receivers.

To make the flow available for other nodes, its metadata is published in a central registry upon initialization (e.g., a master node in a distributed system). For using a flow, sources and targets first need to retrieve the flow metadata from the central registry. The source nodes can then use a flow by pushing tuples into the flow and the target to consume tuples out of the flow by pulling from the flow (see lower part of Figure 1).

In addition to shuffle flows between N senders and M receivers, DFI provides other flow types (i.e., a combiner and a replicate flow) and topologies (i.e., 1:1, N:1, 1:N and N:M) to support various data processing applications. More details about the full programming model of DFI will be explained in Section 4.

3.3 High-level Flow Execution

Key to the execution model of DFI's flows are the design principles discussed above. We achieve these design principles by implementing an execution model where each thread with a source or target has a private send/receive buffer that not only decouples sender from receiver threads but also uses a new memory layout for remote data transfer between sender/receiver threads with only minimal synchronization overhead as we discuss next.

Flow type	*Communication topology*	*Flow options*
Shuffle flow	1:1, N:1, 1:N, N:M	Bandwidth/latency
Replicate flow	1:N, N:M	Bandwidth/latency + ordering guarantees
Combiner flow	N:1	Bandwidth/latency + various aggregations

Table 1: DFI flow types for a wide range of data-centric applications. Communication topologies and flow options further allow applications to adjust the behavior of flows based on application requirements.

In the following, we present the high level execution of flows by following the example of shuffling tuples shown in Figure 1. The push primitive on sources is asynchronous and returns immediately after the tuple to be transferred is copied into the internal send buffer. This non-blocking behavior allows applications to interleave the computation and communication, i.e., pipeline, and thus utilize both CPU and network resources. Moreover, internally the flow execution heavily uses the available one-sided RDMA primitives to reduce the CPU involvement of the targets, and thus decouples the sources and targets as much as possible. To enable one-sided network communication, as mentioned before, a receive buffer must be in place in which the tuples of one or multiple sending threads are written to. Details about the buffer design and their low-overhead synchronization model are discussed further in Section 5.

Once a tuple has been pushed into the flow, a routing decision will be made by the flow based on the provided shuffling key. Depending on the chosen optimization goal (bandwidth or latency), the execution of the flow will transport tuples across the network. For bandwidth optimization, flows batch tuples together destined for the same target in order to achieve a better bandwidth utilization through larger messages. On the other hand if a latency optimization is chosen, the flow execution will prioritize transferring the tuple as soon as possible.

4 Programming Model

In the following, we present a more detailed view on the programming model of DFI and its main abstractions by detailing the opportunities for setting up various communication flow types. In addition, the programming model will be demonstrated through a set of concrete use cases.

4.1 DFI Flows

So far we have only presented a concrete example of constructing a flow for shuffling tuples between a set of source and target threads. However, DFI defines flows with different characteristics to support the wide demands of data processing systems. Table 1 shows the three flow types in DFI, the communication topologies supported by the corresponding flows, as well as their declarative flow options.

The flow abstraction also offers easy adaptability of application algorithms, since different types of flows can be triv-

ially exchanged to offer different behaviors. For instance, to change a symmetric re-partition join algorithm into a fragment-and-replicate join, instead of using a shuffle flow that routes tuples based on the join key, use a replicate flow to replicate the inner table. Performing such algorithmic changes on typical solutions leveraging the RDMA verb interface would infer a significant rewrite of the communication relevant parts of the solution.

In the following, we discuss the different flow types and their potential use in data processing systems.

Shuffle Flow: The shuffle flow is a central abstraction of DFI, where various different communication patterns and routing options can be specified. The communication pattern is indirectly defined by declaring the participating sources and targets in the flow initialization, and can therefore follow 1:1, N:1, 1:N and N:M communication patterns between sending and receiving threads.

The routing of tuples from sources to targets can be defined in three ways in a shuffle flow: (1) The application specifies the shuffle key and let DFI handle the routing. (2) A routing function can be supplied for more control, e.g., to realize different partition functions such as range-partitioning or radix hash partitioning. (3) Lastly, it is also possible to directly specify the node identifier of a target thread on each push into the flow.

Replicate Flow: Another flow type that DFI provides is a so called replicate flow, which targets data processing tasks involving data duplication, such as replicated state machines, fragment-and-replicate join operators or data duplication for stream processing.

The performance of a naïve replication of tuples which uses multiple RDMA operations (i.e., one for each target), will quickly become limited by the outgoing link-speed of the source node; e.g., a replicate flow with 1 source and 8 targets, will have to divide the available network bandwidth at the source, if messages are replicated to all 8 targets on the source node. In DFI, we instead make use of RDMA multicast such that when enabled, messages are replicated in the network as to prevent the outgoing link of the source(s) from becoming a bottleneck.

For some applications using replication, ordering of messages plays an important role. An example of this is state machine replication, where the correctness depends on all replicas processing the incoming operations in the same order. Since many networks (including InfiniBand) do not provide this guarantee if multiple receivers are involved [16] (even not for simple networks with only one switch), replicate flows can be initialized to provide global ordering guarantees, such that all targets consume tuples out of the flow in the same order.

Combiner Flow: The third flow type supported by DFI is the combiner flow. The focus of the combiner flow is many-to-one communication patterns which is typically used in aggregation scenarios, such as a SQL aggregation or a parameter server for distributed machine learning. The combiner flow supports different aggregations (e.g., SUM, COUNT, MIN, MAX) to be performed on the tuples.

Again while a naïve implementation would implement the reduction at the target node, the network can be used to accelerate the reduction. For example, InfiniBand offers the SHARP protocol, that enables in-network aggregations for high-speed InfiniBand networks and thus could help to mitigate when the in-bound network of the receiver becomes a bottleneck.

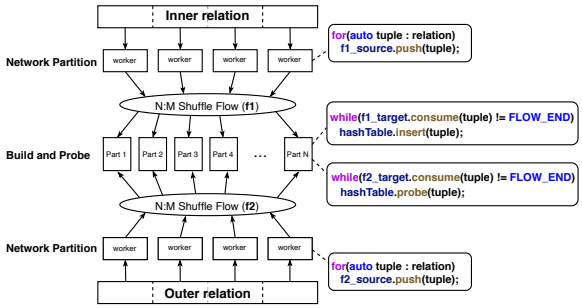


Figure 2: Distributed Radix Hash Join with DFI flows. Two shuffle flows are used to partition tuples across network, one for each relation.

4.2 Use Cases

In the following we present two distributed data processing use cases and how they are realized through DFI: First, we discuss distributed joins for OLAP where the aim is to reduce the runtime by making efficient use of the available network bandwidth. Second, we present a distributed consensus use case where the performance criteria is low latency and high message throughput.

Distributed Radix Join: The distributed radix hash join is a popular join operator due to its dominating performance [3, 2]. The idea behind the radix hash join is to partition the input relations into such small partitions that the resulting hash tables fit into the CPU caches to reduce cache-misses.

In its original form the distributed radix join has a high level of complexity since multiple senders need to coordinate when writing to the same receivers. For example, in [3, 2], histograms of buckets are pre-computed in a first pass on each input table to allocate private memory buffers for each thread on the receiver node and then use coordination-free one-sided communication in a second pass to shuffle the data of each input table.

We argue that with DFI, the design of a distributed radix join is simpler while the performance is on par (and sometimes even better) with the latest distributed radix join implementations (as will be shown in Section 6.2). To realize the join with DFI, two bandwidth optimized shuffle flows are used as shown in Figure 2, one for shuffling each relation. Figure 2 also shows the pseudo-code how tuples can be pushed into the flows during network partitioning, and consumed at the target (i.e., receiver node) out of the flows for the relations to either build the hash table (for the inner relation) or probe the hash table (for the outer relation).

The shuffle flows for the join are initialized with one source per sender thread and one target per output partition. That way the flow can used for achieving the desired partition fan-out. The routing of tuples to the partition-specific *targets* is done on a per thread level by passing a radix hash function to DFI as the routing function. This also leads to a reduction of complexity of the DFI join compared to the original RDMA-based distributed radix join since the histogram computation can be omitted. Moreover, the memory management of local and remote buffers is handled in DFI.

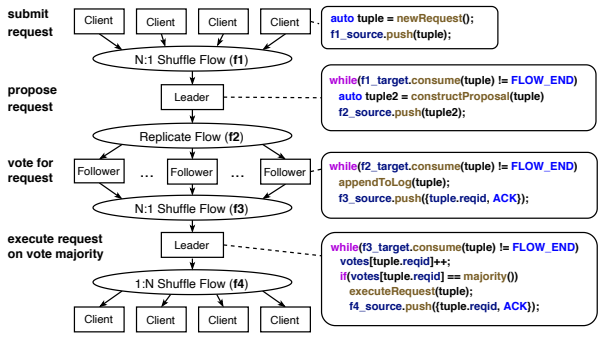


Figure 3: Leader-based consensus with DFI flows. Four flows are used to realize consensus between replicas.

Distributed Consensus: Consensus in a distributed system describes the agreement of multiple (often asynchronous) participants on a single value, or a sequence of values, while tolerating the presence of faulty participants. It is a fundamental primitive in distributed computing which is needed, for example, for the reliable implementation of replicated state machines, leader election, or system reconfiguration.

Classical consensus protocols [14, 18] are centered around a centralized coordinator, called leader. The leader orders concurrently arriving requests of participants (i.e., clients) and forwards them to a set of so called followers. The followers vote for requests that they receive from the leader. Once the leader has received a majority of votes (itself included), the leader can notify the corresponding client that its request was agreed-upon. The high-level message flow of a leader-based consensus implementation using DFI can be modeled directly with the flows provided by DFI and is depicted in Figure 3. Figure 3 additionally shows pseudo-code of how these flows are used for the communication which we explain in the following.

Clients initially send their vote with an N:1 shuffle flow to the leader. The replicate flow is ideal to handle the communication from the leader to its followers, as all followers receive identical messages. The use of the RDMA multicast verbs built into DFI alleviates load placed on the leader compared to the naïve replication of messages. This is an interesting optimization, as the leader is typically a major bottleneck in consensus-based systems. Once followers received the request and voted for a result, they send the outcome back to the leader, again using a shuffle flow. In a last step the leader distributes the consensus-outcome to the client using the client IDs as the shuffle key.

An interesting optimization that DFI provides is to use the optimization option for global ordered multicast (also referred to as ordered unreliable multicast - OUM). In particular, Li et al. [16] propose a single round-trip consensus protocol based on OUM. While this work focuses on Ethernet-based systems, to our knowledge, DFI is the first system that can provide these semantics in the context of InfiniBand.

As we show in Section 6.2.2, using the ordered multicast significantly improves both throughput and latency compared to conventional consensus protocol designs using native RDMA that follow more classical consensus designs.

5 Flow Implementation

In this section, we briefly present how we realized the key design principles discussed in Section 3 through an implementation of DFI. A more extensive discussion of the implementation along with details on the different flow implementations can be found in the original publication [20].

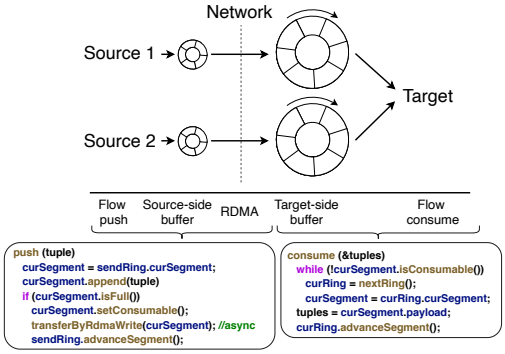


Figure 4: DFI flow implementation using ring buffers. In DFI flows, each source allocates a private target-side ring buffer to minimize coordination overhead.

5.1 Flow Execution

The key design principles listed in Section 3 impose challenges for how the data transfer between the sources and targets is realized which are pivotal for distributed data processing. In the following, we give a brief overview of the flow execution paired with the design principles.

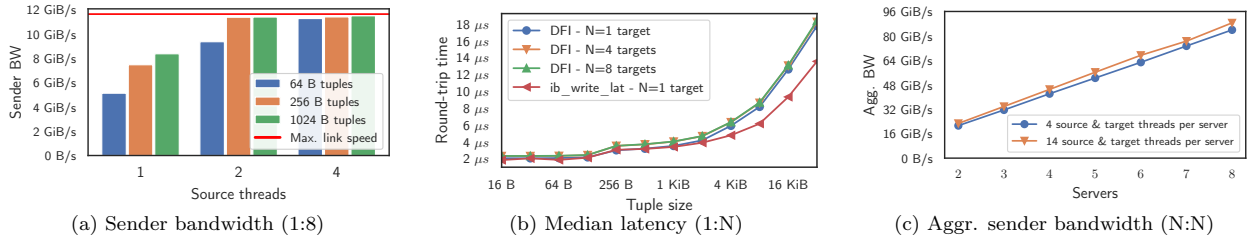
On a high-level, DFI uses a private send/receive buffer for each pair of source and target threads as illustrated in Figure 4. The design of source- and target-side buffers follows a ring-based design where each ring is composed of a configurable number of segments and is allocated as one consecutive RDMA-enabled region in memory. The segment itself can be sized to contain a single tuple up to a batch of tuples. Therefore, the segment size is a tuning parameter that allows DFI to either optimize for bandwidth or latency independent of the tuple sizes used by the application.

One key question is how such a segmented ring design enables pipelining of tuples with low-overhead synchronization. By having private send/receive buffers for each pair of sources and targets, no synchronization or coordination is needed between, e.g., multiple sources writing to the same target. As such DFI can effectively scale-out to many sources and targets without negatively impacting the performance. In order to achieve pipelined data transfer between buffers (i.e., a decoupling of senders and receivers), one-sided RDMA writes are used to copy data asynchronously from sources to targets. This asynchronous data transfer using RDMA writes is implemented by the `transferByRdmaWrite` call in Figure 4. This method also implements the synchronization with the target buffer to not overwrite any segments that has not been consumed yet. The synchronization is based on a credit approach where sources only have to synchronize with target buffers once the credit is used up.

6 Experimental Evaluation

We now evaluate DFI by first looking at the efficiency of DFI in terms of how well the high-level interface utilizes the network compared to low-level RDMA verbs. Subsequently we look at two typical use cases in data processing systems and compare the implementations to existing state-of-the-art solutions.

In all experiments we use the notation (N:M) to indicate the number of servers involved in a flow topology. The num-



(a) Sender bandwidth (1:8) (b) Median latency (1:N) (c) Aggr. sender bandwidth (N:N)

Figure 5: Shuffle flow performance. DFI achieves max. bandwidth and low latency for various scenarios.

ber of threads per server is reported separately per experiment.

Evaluation Environment: All experiments were conducted on an 8 node cluster where 6 of the nodes are equipped with two Intel(R) Xeon(R) Gold 5120 CPUs (14 cores) and 512 GB main-memory, and 2 nodes equipped with two Intel(R) Xeon(R) Gold 5220 CPUs (18 cores). Hyper-threading is disabled for all nodes. Each node is equipped with two Mellanox ConnectX-5 MT27800 NICs (InfiniBand EDR 4x NICs, 100 Gbps), connected to one SB7890 InfiniBand switch. The operating system is Ubuntu 18.04.1 LTS, with Linux 4.15.0-47 kernel on all nodes. DFI is implemented with C++17 and compiled with gcc-7.3.0.

6.1 Experiment 1: Efficiency of DFI

The first experiment shows the efficiency of DFI compared to low-level RDMA verbs. In the following, we evaluate the shuffle flows with bandwidth and latency optimization and lastly, a scale-out experiment is presented. For an extensive evaluation of the flows available in DFI, see full paper version [20].

Bandwidth-Optimized: Our first experiment evaluates performance for the shuffle flow from 1 server to 8 servers with varying tuple sizes. Further, we vary the number of sources (threads) pushing tuples into the flow. The batch size for the bandwidth optimized version in our experiments is 8 KiB. We choose a batch size of 8 KiB as this offers a good trade-off between network bandwidth and time until the batch is filled.

Figure 5a reports results for the bandwidth-optimized flow. As we see, in most settings we achieve the full network bandwidth. Only, the single-threaded scenario shows some overhead since batches must first be filled on the source side with individual tuples before they can be transferred to the target. This overhead can, however, be amortized by using more threads per server as shown in Figure 5a. Due to the efficient multi-threading support of DFI, we see that from two source threads on, the bandwidth is limited by the speed of the outgoing link (100 Gbps / 11,64 GiB/s - red line) for tuple sizes larger than 128 B. Moreover, when using 4 threads the maximal bandwidth is achieved independent of tuple sizes.

Latency-Optimized: We additionally evaluated the shuffle flow that implements latency optimizations. For measuring latency, two shuffle flows are used to implement a request and response pattern to measure the round-trip time between two nodes. To show that DFI's buffer design only adds minimal latency overhead, we compare the latency of DFI to *ib_write_lat*[2] which is a standard tool for performance testing that uses low-level verbs to implement a round-trip between a sender and a receiver node. For DFI, we additionally used a varying number of receiving servers (1, 4, and 8) to observe the effect on latency when shuffling to various destinations.

As we see in Figure 5b, the median latency of DFI for one full round-trip only adds minimal overhead when compared to *ib_write_lat* which is due to the intermediate copy to the buffer. Moreover, keep in mind that DFI provides a high-level abstraction and thus not only reduces application complexity but also provides several optimizations to applications. This includes an efficient overlapping of compute and communication as well as many other optimizations such as efficient replication and ordering guarantees. As we show in Section 6.2, this enables DFI to provide superior performance in different use cases when compared to existing approaches that are using other interfaces (low-level RDMA verbs or MPI).

Moreover, the advantage of DFI compared to plain RDMA is the encapsulated memory management, which allows applications to use RDMA transparently without hand-tuned memory management while still achieving optimal performance. The experiment shows that this abstraction hardly incurs any overhead compared to *ib_write_lat*. For multiple targets the latency of DFI is only slightly higher due to the internal routing in the shuffle flow. Multiple targets are not supported by *ib_write_lat* though (i.e., *ib_write_lat* uses only one target in this experiment).

Scale-out: Since data processing systems often need to scale out to many nodes, we conducted a scale-out experiment for the shuffle flow, increasing the number of source and target servers. Moreover, we use 14 sources and targets on all nodes which in total gives 12544 unique source/target connections for the maximal number of nodes used. As shown in Figure 5c, DFI scales linearly with the number of nodes (as indicated by the x-axis), effectively increasing the aggregated bandwidth with the link-speed of each added node.

Key Insights (Exp. 1): Our results show, that DFI flows can provide a high-level abstraction with no or only negligible overhead compared to low-level RDMA verbs.

6.2 Experiment 2: Use Cases

We now evaluate DFI by implementing the two use cases we discussed in Section 4.2.

6.2.1 *Distributed Joins*

Distributed joins are crucial operators in OLAP due to large amounts of data having to be transferred across the network, and therefore a good candidate to evaluate bandwidth-optimized flows of DFI.

[2] `https://github.com/linux-rdma/perftest`.

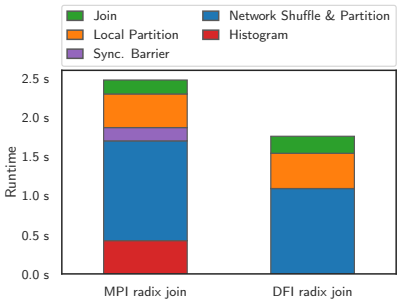


Figure 6: Dist. radix join - 8 nodes, 64 threads (DFI)/64 processes (MPI) in total. 2.56 B ⋈ 2.56 B tuples.

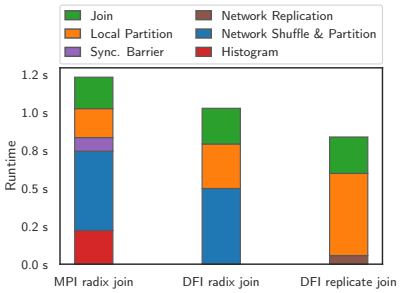


Figure 7: Dist. joins - 8 nodes, 64 threads (DFI)/64 processes (MPI) in total. 2.56 M ⋈ 2.56 B tuples.

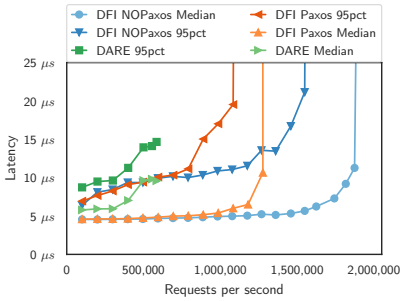


Figure 8: Performance comparison of DARE [19] with DFI-based implementations of Multi-Paxos and NOPaxos.

Radix Join: We implemented a distributed radix hash join on DFI and compared its performance to a state-of-the-art implementation for RDMA using MPI [2]. Both implementations employ the same optimizations (e.g., write-combine buffer in partitioning phase and tuple compression). However, the MPI join of [2] uses multi-process parallelism while our join uses multi-threading instead. Figure 6 shows the average runtime of the two joins for all 8 nodes.

The DFI radix join achieves the best runtime mainly due to two design choices of DFI. At first, the DFI radix join does not need to first compute a global histogram of the partition buckets. The MPI radix join in [2] makes use of one-sided *MPI_Put* primitives. In order to achieve coordination free writes, it thus has to compute exclusive writing offsets for each partition using one additional pass. Different from this, DFI encapsulates the memory management through our buffer design which makes the additional pass superfluous.

The other reason for the runtime gap is due to the synchronization barrier needed in the MPI radix join after the network partition phase. Here, the join algorithm needs to make sure that all data has arrived before starting to process the local partitioning. While the data in this experiment is uniformly distributed, some runtime variance between multiple parallel workers still exists and is more pronounced in high-speed networks. This synchronization is not needed with DFI, since incoming tuples can already be processed when they arrive in a streaming-wise fashion.

Join Adaptability: Flows in DFI offer a high-level abstraction which encapsulates the data transfer of applications. As a result, it is trivial to adapt algorithms to use a different communication pattern. To demonstrate this, we adapted our radix hash join implementation to a fragment-and-replicate join variant which uses one replicate flow that replicates the inner table on all nodes. Figure 7 shows the runtimes of the three different join implementations with a smaller inner table (1000× smaller than the outer table). The replication of the small inner table is comparably cheap compared to shuffling the big outer table over the network. Overall, for this setup this helps to further reduce the overall runtime by another 20%,

6.2.2 State Machine Replication

In this experiment, we implemented a simple key-value store that replicates data using a consensus protocol. For the experiment, we used two different consensus protocols, classical Multi-Paxos [14] and NOPaxos [16]. We modeled the normal, failure-free operation of Multi-Paxos as depicted in Figure 3. For NOPaxos, we implemented its *normal operation* protocol, which relies on the OUM primitive that can be provided by DFI's replicate flow, as well as its *gap agreement* protocol to detect lost messages. We compare both implementations with DARE [19], a state-of-the-art replicated key-value store that is based on a hand-crafted consensus protocol and heavily relies on one-sided RDMA.

We deployed all approaches with five replicas (a leader and four followers). Load was generated by six clients distributed across three separate nodes. Clients submitted 64 byte sized requests using YCSB's read-dominated workload [6] (95% reads and 5% writes). The results are shown in Figure 8.

The two DFI-based implementations consistently outperform DARE in our settings in both achieved throughput and latency. This is caused mainly by DARE's sequential design. First, each DARE client cannot submit a new request until it has received the result from its previous request, which limits its achievable throughput.

Second, DARE's write protocol serializes requests. While this limitation is mitigated by separately batching reads and writes, a mix of both request types frequently interrupts batches [21]. This is confirmed by DARE's own evaluation [19].

Our Multi-Paxos and NOPaxos implementation exhibit near-identical response latencies as long as they are not saturated. This appears counter-intuitive at first, as Multi-Paxos requires four message delays to respond to a client, whereas two messages delays suffices for NOPaxos as long as no messages are lost. However, fetching a global sequence number from the tuple sequencer of the ordered replicate flow incurs an additional two message delays.

For a load higher than 700k requests/s, we see benefits of our NOPaxos over our Multi-Paxos implementation. Under this load, the leader in Multi-Paxos becomes saturated as it has to repeatedly collect responses from a majority of replicas. In contrast, in NOPaxos the clients themselves collect these responses. This alleviates the burden placed on the leader in Multi-Paxos, which leads to stable response latencies in DFI's NOPaxos up to even higher request rates of almost 1.5M (95th percentile).

Key Insights (Exp. 2): In summary, DFI does not only achieve a better performance for distributed joins and consensus than state-of-the-art, but also offers an ease-of-use high-level abstraction to implement efficient solutions with a low code complexity.

7 Using DFI

We provide DFI as an open-source project.[3] In the repository, guides can be found detailing how to set up and include DFI, either as a git submodule or installation, along with small examples showing the setup and usage of DFI flows.

By open-sourcing our implementation, we hope to stimulate not only follow-up research but also any form of contributions and invite any interested parties to collaborate and build DFI further. Additionally we hope that commercial vendors will contribute and provide a DFI implementation also for other high-speed network stacks, similar to the collaborative effort of MPI.

8 Conclusions

In this paper, we presented DFI, a new data-centric interface for fast networks. With our implementation for InfiniBand we have shown that DFI adds only minor overhead compared to low-level abstractions such as RDMA verbs. Moreover, by implementing two use cases, we demonstrated that DFI can efficiently support data-centric applications with different requirements (high-bandwidth vs. low-latency) at high performance.

In future, we plan to integrate further useful extensions into DFI flows such as fault-tolerance as well as elasticity of flows to add/remove nodes at runtime. Additionally, we aim to extend DFI flows to support other compute architectures such as GPUs and DPUs.

9 Acknowledgments

This work was partially funded by the German Research Foundation (DFG) under the grants BI2011/1 & BI2011/2 (DFG priority program 2037), the DFG Collaborative Research Center 1053 (MAKI) as well as gifts from Mellanox and Huawei.

10 References

[1] G. Alonso, C. Binnig, I. Pandis, K. Salem, J. Skrzypczak, R. Stutsman, L. Thostrup, T. Wang, Z. Wang, and T. Ziegler. DPI: the data processing interface for modern networks. In *CIDR*, 2019.

[2] C. Barthels et al. Distributed join algorithms on thousands of cores. *PVLDB*, 10(5):517–528, 2017.

[3] C. Barthels, S. Loesing, G. Alonso, and D. Kossmann. Rack-scale in-memory join processing using rdma. In *SIGMOD*, page 14631475, 2015.

[4] 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.

[5] Q. Cai, W. Guo, H. Zhang, D. Agrawal, G. Chen, B. C. Ooi, K. Tan, Y. M. Teo, and S. Wang. Efficient distributed memory management with RDMA and caching. *PVLDB*, 11(11):1604–1617, 2018.

[6] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *SoCC*, pages 143–154, 2010.

[7] A. Dragojevic, D. Narayanan, M. Castro, and O. Hodson. Farm: Fast remote memory. In R. Mahajan and I. Stoica, editors, *NSDI*, pages 401–414, 2014.

[8] P. Fent, A. van Renen, A. Kipf, V. Leis, T. Neumann, and A. Kemper. Low-latency communication for fast DBMS using RDMA and shared memory. In *ICDE*, pages 1477–1488, 2020.

[9] P. W. Frey and G. Alonso. Minimizing the hidden cost of RDMA. In *ICDCS*, pages 553–560, 2009.

[10] W. Gropp et al. *Using Advanced MPI: Modern Features of the Message-Passing Interface*. The MIT Press, 2014.

[11] A. Kalia, M. Kaminsky, and D. G. Andersen. Using RDMA efficiently for key-value services. In *SIGCOMM*, pages 295–306, 2014.

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

[13] S. J. Kang, S. Y. Lee, and K. M. Lee. Performance comparison of openmp, mpi, and mapreduce in practical problems. *Advances in Multimedia*, 2015.

[14] L. Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[15] B. Li, Z. Ruan, W. Xiao, Y. Lu, Y. Xiong, A. Putnam, E. Chen, and L. Zhang. Kv-direct: high-performance in-memory key-value store with programmable NIC. In *SOSP*, pages 137–152, 2017.

[16] J. Li, E. Michael, N. K. Sharma, A. Szekeres, and D. R. K. Ports. Just say NO to paxos overhead: Replacing consensus with network ordering. In *OSDI*, pages 467–483, 2016.

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

[18] D. Ongaro and J. K. Ousterhout. In search of an understandable consensus algorithm. In *ATC*, pages 305–319, 2014.

[19] M. Poke and T. Hoefler. DARE: high-performance state machine replication on RDMA networks. In *HPDC*, pages 107–118, 2015.

[20] L. Thostrup, J. Skrzypczak, M. Jasny, T. Ziegler, and C. Binnig. DFI: the data flow interface for high-speed networks. In *SIGMOD*, pages 1825–1837, 2021.

[21] C. Wang, J. Jiang, X. Chen, N. Yi, and H. Cui. APUS: fast and scalable paxos on RDMA. In *SoCC*, pages 94–107, 2017.

[22] J. Xue, Y. Miao, C. Chen, M. Wu, L. Zhang, and L. Zhou. Fast distributed deep learning over RDMA. In *EuroSys*, pages 44:1–44:14, 2019.

[23] E. Zamanian, J. Shun, C. Binnig, and T. Kraska. Chiller: Contention-centric transaction execution and data partitioning for modern networks. In *SIGMOD*, page 511526, 2020.

[24] T. Ziegler, V. Leis, and C. Binnig. Rdma communciation patterns. *Datenbank-Spektrum*, 20(3):199–210, Nov 2020.

[25] T. Ziegler, S. Tumkur Vani, C. Binnig, R. Fonseca, and T. Kraska. Designing distributed tree-based index structures for fast rdma-capable networks. In *SIGMOD*, page 741758, 2019.

[3] `https://github.com/DataManagementLab/DFI-public`

Technical Perspective: FoundationDB: A Distributed Unbundled Transactional Key Value Store

Alfons Kemper
Technical University of Munich

With the emergence of (geographically) distributed data mangement in cloud infrastructures the key value systems were promoted as so-called NoSQL systems. In order to achieve maximum availability and performance these KV stores sacrificed the "holy grail" of database consistency and relied on relaxed consistency models, such as *eventual consistency*. This was a consequence of Eric Brewer's so-called CAP observation (aka Theorem) [1] stating that only two of the three desiderata of distributed systems could possibly be satisfied at the same time: (1) Consistency, where every read operation receives the most recent committed write, (2) Availability which nowadays typically strives for the so-called many (e.g., seven or even nine) nines meaning 99.99999% up-time, (3) Partition tolerance which demands that the system has to remain operational even under severe network malfunctions. As a consequnce, NoSQL system designers traded consistency for higher availability and network fault tolerance. The relaxed replica convergence models were categorized by Werner Vogels [2] of Amazon in 2009. However, many mission critical applications even in cloud settings do require stronger consistency guarantees. Eventual consistency only ensures convergence to the same state of all replicas. FoundationDB, on the other hand, is a scalable distributed key value store with strong consistency guarantees. It started 10 years ago as an open source project and is now widely used as a mission-critical backbone repository in cloud infrastructures, such as Apple and Snowflake. In this respect FoundationDB re-unites the *NoSQL* paradigm of high availability and low latency with the ACID (Atomicity, Consistency, Isolation, Durability) guarantees imposed by traditional database systems. This new breed of systems is therefore coined *NewSQL* – albeit not all offering SQL interfaces. For scalability and elasticity in cloud infrastructures FoundationDB exhibits a fully disaggregated architecture consisting of a storage system (SS), a logging system (LS), and a separated transaction system (TS). Storage Servers are decoupled from Log Servers, which maintain the "ground truth". Storage Servers continuously apply log records from Log Servers in order to lag only slightly behind the transaction commit state – in production system measurements the authors report a lag in the order of just milliseconds only. The transaction system employs an optimistic multi version concurrency control (MVCC) scheme with a subsequent verification phase to ensure strict serializability. Thereby, read only transactions are not penalized as they access versions that were committed by the time they started. For this purpose a so-called Sequencer process determines the corresponding time stamp that is then observed by proxies to access the correct version from storage servers. For write transactions, the Sequencer assigns the commit time stamp with which the resolvers verify strict serializability by comparing the transaction's read set (i.e., the key ranges of key value pairs that were accessed in the course of the transaction) with the write sets of transactions committed in the mean time. Successfully verified transactions are made durable by writing their logs to multiple log servers for fault tolerance. These log servers can be replicated across distinct (geographic) regions to tolerate failures (such as power outages) within an entire region. The FoundationDB system can be configured for synchronous as well as asynchronous log transferral – thereby trading off between safety and latency requirements. In case of a primary server failure the synchronous mode guarantees seamless instantaneous takeover by the secondary server – albeit incurring extra latency overhead during commit processing. Under asynchronous mode the primary server maintains several satellites within the same region which, in case of failure, can submit the log's suffix that was not yet sent to the secondary server in a remote region. Thus, the tradeoff between performance/responsiveness and failure resilience is one of the foremost goals of the FoundationDB design. Correctness and Robustness was also a key aspect in the development life cycle of the FoundationDB system. In order to achieve this the team relied on simulation testing that allowed to run the distributed software code in a deterministic manner for error reproducibility and extensive test coverage.

To conclude, the design of FoundationDB lays the "foundation" for future cloud information systems that have to balance performance and consistency requirements.

1. REFERENCES

[1] Armando Fox and Eric Brewer. Harvest, Yield and Scalable Tolerant Systems. In *Proc. 7th Workshop Hot Topics in Operating Systems (HotOS 99)*, IEEE CS, 1999, pg. 174–178.

[2] Werner Vogels. Eventually consistent. *Communications of the ACM*. 52: 40–44, 2009.

FoundationDB: A Distributed Key Value Store

Jingyu Zhou*, Meng Xu*, Alexander Shraer†, Bala Namasivayam*, Alex Miller‡,
Evan Tschannen‡, Steve Atherton‡, Andrew J. Beamon‡, Rusty Sears*, John Leach*,
Dave Rosenthal*, Xin Dong*, Will Wilson◇, Ben Collins◇, David Scherer◇, Alec Grieser*,
Young Liu†, Alvin Moore*, Bhaskar Muppana*, Xiaoge Su*, Vishesh Yadav*
*Apple Inc. ‡Snowflake Inc. † Cockroach Labs ◇antithesis.com

ABSTRACT

FoundationDB is an open source transactional key value store created more than ten years ago. It is one of the first systems to combine the flexibility and scalability of NoSQL architectures with the power of ACID transactions. FoundationDB adopts an unbundled architecture that decouples an in-memory transaction management system, a distributed storage system, and a built-in distributed configuration system. Each sub-system can be independently provisioned and configured to achieve scalability, high-availability and fault tolerance. FoundationDB includes a deterministic simulation framework, used to test every new feature under a myriad of possible faults. FoundationDB offers a minimal and carefully chosen feature set, which has enabled a range of disparate systems to be built as layers on top. FoundationDB is the underpinning of cloud infrastructure at Apple, Snowflake and other companies.

1 Introduction

Many cloud services rely on scalable, distributed storage backends for persisting application state. Such storage systems must be fault tolerant and highly available, and at the same time provide sufficiently strong semantics and flexible data models to enable rapid application development. Such services must scale to billions of users, petabytes or exabytes of stored data, and millions of requests per second.

More than a decade ago, NoSQL storage systems emerged offering ease of application development, making it simple to scale and operate storage systems, offering fault-tolerance and supporting a wide range of data models (instead of the traditional rigid relational model). In order to scale, these systems sacrificed transactional semantics, and instead provided eventual consistency, forcing application developers to reason about interleavings of updates from concurrent operations.

©ACM 2022. This is a minor revision of the paper entitled "FoundationDB: A Distributed Unbundled Transactional Key Value Store", published in SIGMOD '21, June 20–25, 2021, Virtual Event, China. DOI: https://doi.org/10.1145/3448016.3457559

FoundationDB (FDB) [4] was created in 2009 and gets its name from the focus on providing what we saw as the foundational set of building blocks required to build higher-level distributed systems. It is an ordered, transactional, key-value store natively supporting multi-key strictly serializable transactions across its entire key-space. Unlike most databases, which bundle together a storage engine, data model, and query language, forcing users to choose all three or none, FDB takes a modular approach: it provides a highly scalable, transactional storage engine with a minimal yet carefully chosen set of features. The NoSQL model leaves application developers with great flexibility. Applications can manage data stored as simple key-value pairs, but at the same time implement advanced features, such as consistent secondary indices and referential integrity checks [16]. FDB defaults to strictly serializable transactions, but allows relaxing these semantics for applications that don't require them with flexible, fine-grained controls over conflicts.

One of the reasons for its popularity and growing open source community is FoundationDB's focus on the "lower half" of a database, leaving the rest to its "layers"—stateless applications developed on top to provide various data models and other capabilities. With this, applications that would traditionally require completely different types of storage systems, can instead all leverage FDB. Indeed, the wide range of layers that have been built on FDB in recent years are evidence to the usefulness of this unusual design. For example, the FoundationDB Record Layer [16] adds back much of what users expect from a relational database, and JanusGraph [8], a graph database, provides an implementation as a FoundationDB layer [7]. In its newest release, CouchDB [1] (arguably the first NoSQL system) is being re-built as a layer on top of FoundationDB.

Testing and debugging distributed systems is at least as hard as building them. Unexpected process and network failures, message reorderings, and other sources of non determinism can expose subtle bugs and implicit assumptions that break in reality, which are extremely difficult to reproduce or debug. The consequences of such subtle bugs are especially severe for database systems, which purport to offer perfect fidelity to an unambiguous contract. Moreover, the stateful nature of a database system means that any such bug can result in subtle data corruption that may not be discovered for months. FDB took a radical approach—before building the database itself, we built a deterministic database simulation framework that can simulate a network of interacting processes and a variety of disk, process, network, and request-level failures and recoveries, all within a

single physical process. This rigorous testing in simulation makes FDB extremely stable, and allows its developers to introduce new features and releases in a rapid cadence.

FDB adopts an unbundled architecture [29] that comprises a control plane and a data plane. The control plane manages the metadata of the cluster and uses Active Disk Paxos [15] for high availability. The data plane consists of a transaction management system, responsible for processing updates, and a distributed storage layer serving reads; both can be independently scaled out. FDB achieves strict serializability through a combination of optimistic concurrency control (OCC) [25] and multi-version concurrency control (MVCC) [12]. One of the features distinguishing FDB from other distributed databases is its approach to handling failures. Unlike most similar systems, FDB does not rely on quorums to mask failures, but rather tries to eagerly detect and recover from them by reconfiguring the system. This allows us to achieve the same level of fault tolerance with significantly fewer resources: FDB can tolerate f failures with only $f + 1$ (rather than $2f + 1$) replicas.

This paper makes three primary contributions. First, we describe an open source distributed storage system, FoundationDB, combining NoSQL and ACID, used in production at Apple, Snowflake, and other companies. Second, an integrated deterministic simulation framework makes FoundationDB one of the most stable systems of its kind. Third, we describe a unique architecture and approach to transaction processing, fault tolerance, and high availability.

2 Design

The main design principles of FDB are:

- *Divide-and-Conquer (or separation of concerns).* FDB decouples the transaction management system (write path) from the distributed storage (read path) and scales them independently. Within the transaction management system, processes are assigned various roles representing different aspects of transaction management. Furthermore, cluster-wide orchestrating tasks, such as overload control, and load balancing are also divided and serviced by additional heterogeneous roles.
- *Make failure a common case.* For distributed systems, failure is a norm rather than an exception. To cope with failures in the transaction management system of FDB, we handle all failures through the recovery path: the transaction system proactively shuts down when it detects a failure. Thus, all failure handling is reduced to a single recovery operation, which becomes a common and well-tested code path. To improve availability, FDB strives to minimize Mean-Time-To-Recovery (MTTR). In our production clusters, the total time is usually less than five seconds.
- *Simulation testing.* FDB relies on a randomized, deterministic simulation framework for testing the correctness of its distributed database. Simulation tests not only expose deep bugs [27], but also boost developer productivity and the code quality of FDB.

2.1 Architecture

An FDB cluster has a control plane for managing critical system metadata and cluster-wide orchestration, and a data plane for transaction processing and data storage, as illustrated in Figure 1.

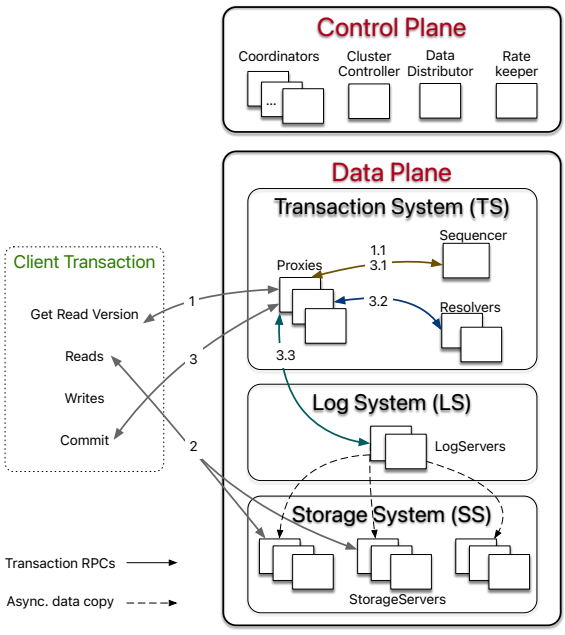


Figure 1: Architecture and the transaction processing.

2.1.1 Control Plane

The control plane is responsible for persisting critical system metadata, i.e., the configuration of transaction systems, on `Coordinators`. These `Coordinators` form a Paxos group [15] and elect a `ClusterController`. The `ClusterController` monitors all servers in the cluster and recruits three processes, `Sequencer` (described in Section 2.1.2), `DataDistributor`, and `Ratekeeper`, which are re-recruited if they fail or crash. The `DataDistributor` is responsible for monitoring failures and balancing data among `StorageServers`. `Ratekeeper` provides overload protection for the cluster.

2.1.2 Data Plane

FDB targets OLTP workloads that are read-mostly, read and write a small set of keys per transaction, have low contention, and require scalability. FDB chooses an unbundled architecture [29]: a distributed transaction management system (TS) consists of a `Sequencer`, `Proxies`, and `Resolvers`, all of which are stateless processes. A log system (LS) stores Write-Ahead-Log (WAL) for TS, and a separate distributed storage system (SS) is used for storing data and servicing reads. The LS contains a set of `LogServers` and the SS has a number of `StorageServers`. This scales well to Apple's largest transactional workloads [16].

The `Sequencer` assigns a read and a commit version to each transaction and, for historical reasons, recruits `Proxies`, `Resolvers`, and `LogServers`. `Proxies` offer MVCC read versions to clients and orchestrate transaction commits. `Resolvers` check for conflicts among transactions. `LogServers` act as replicated, sharded, distributed persistent queues, each queue storing WAL data for a `StorageServer`.

The SS consists of a number of `StorageServers`, each storing a set of data shards, i.e., contiguous key ranges, and serving client reads. `StorageServers` are the majority of processes in the system, and together they form a distributed B-tree. Currently, the storage engine on each `StorageServer` is an enhanced version of SQLite [24].

2.1.3 Read-Write Separation and Scaling

As mentioned above, processes are assigned different roles; FDB scales by adding new processes for each role. Clients read from sharded `StorageServers`, so reads scale linearly with the number of `StorageServers`. Writes are scaled by adding more `Proxies`, `Resolvers`, and `LogServers`. The control plane's singleton processes (e.g., `ClusterController` and `Sequencer`) and `Coordinators` are not performance bottlenecks; they only perform limited metadata operations.

2.1.4 Bootstrapping

FDB has no dependency on external coordination services. All user data and most system metadata (keys that start with `0xFF` prefix) are stored in `StorageServers`. The metadata about `StorageServers` is persisted in `LogServers`, and the `LogServers` configuration data is stored in all `Coordinators`. The `Coordinators` are a disk Paxos group; servers attempt to become the `ClusterController` if one does not exist. A newly elected `ClusterController` recruits a new `Sequencer`, which reads the old LS configuration from the `Coordinators` and spawns a new TS and LS. `Proxies` recover system metadata from the old LS, including information about all `StorageServers`. The `Sequencer` waits until the new TS finishes recovery (Section 2.2.4), then writes the new LS configuration to all `Coordinators`. The new transaction system is then ready to accept client transactions.

2.1.5 Reconfiguration

The `Sequencer` process monitors the health of `Proxies`, `Resolvers`, and `LogServers`. Whenever there is a failure in the TS or LS, or the database configuration changes, the `Sequencer` terminates. The `ClusterController` treats this as a `Sequencer` failure and recruits a new `Sequencer`, which follows the above bootstrapping process to spawn a new TS and LS. In this way, transaction processing is divided into epochs, where each epoch represents a generation of the transaction management system with its own `Sequencer`.

2.2 Transaction Management

This section describes end-to-end transaction processing and strict serializability, then discusses logging and recovery.

2.2.1 End-to-end Transaction Processing

As illustrated in Figure 1, a client transaction starts by contacting one of the `Proxies` to obtain a read version (i.e., a timestamp). The `Proxy` then asks the `Sequencer` for a read version that at least as large as all previously issued transaction commit versions, and sends this read version back to the client. The client may then issue reads to `StorageServers` and obtain values at that specific read version. Client writes are buffered locally without contacting the cluster and read-your-write semantics are preserved by combining results from database look-ups with uncommitted writes of the transaction. At commit time, the client sends the transaction data, including the read and write sets (i.e., key ranges), to one of the `Proxies` and waits for a commit or abort response. If the transaction cannot commit, the client may choose to restart it.

A `Proxy` commits a client transaction in three steps. First, it contacts the `Sequencer` to obtain a commit version that is larger than any existing read versions or commit versions. The `Sequencer` chooses the commit version by advancing it at a rate of one million versions per second. Then, the `Proxy` sends the transaction information to range-partitioned `Resolvers`, which implement FDB's optimistic concurrency control by checking for *read-write* conflicts. If all `Resolvers` return with no conflict, the transaction can proceed to the final commit stage. Otherwise, the `Proxy` marks the transaction as aborted. Finally, committed transactions are sent to a set of `LogServers` for persistence. A transaction is considered committed after all designated `LogServers` have replied to the `Proxy`, which reports the committed version to the `Sequencer` (to ensure that later transactions' read versions are after this commit) and then replies to the client. `StorageServers` continuously pull mutation logs from `LogServers` and apply committed updates to disks.

In addition to the above *read-write transactions*, FDB also supports *read-only transactions* and *snapshot reads*. A read-only transaction in FDB is both serializable (happens at the read version) and performant (thanks to the MVCC), and the client can commit these transactions locally without contacting the database. This is particularly important because the majority of transactions are read-only. Snapshot reads in FDB selectively relax the isolation property of a transaction by reducing conflicts, i.e., concurrent writes will not conflict with snapshot reads.

2.2.2 Strict Serializability

FDB implements Serializable Snapshot Isolation (SSI) by combining OCC with MVCC. Recall that a transaction T_x gets both its read version and commit version from the `Sequencer`, where the read version is guaranteed to be no less than any committed version when T_x starts and the commit version is larger than any existing read or commit versions. This commit version defines a serial history for transactions and serves as a Log Sequence Number (LSN). Because T_x observes the results of all previous committed transactions, FDB achieves strict serializability. To ensure there are no gaps between LSNs, the `Sequencer` returns the previous commit version (i.e., previous LSN) with each commit version. A `Proxy` sends both LSN and the previous LSN to `Resolvers` and `LogServers` so that they can serially process transactions in the order of LSNs. Similarly, `StorageServers` pull log data from `LogServers` in increasing LSN order.

`Resolvers` use a lock-free conflict detection algorithm similar to *write-snapshot isolation* [34], with the difference that in FDB the commit version is chosen before conflict detection. This allows FDB to efficiently batch-process both version assignments and conflict detection.

The entire key space is divided among `Resolvers` allowing conflict detection to be performed in parallel. A transaction can commit only when all `Resolvers` admit the transaction. Otherwise, the transaction is aborted. It is possible that an aborted transaction is admitted by a subset of `Resolvers`, and they have already updated their history of potentially committed transactions, which may cause other transactions to conflict (i.e., a false positive). In practice, this has not been an issue for our production workloads, because transactions' key ranges usually fall into one `Resolver`. Additionally, because the modified keys expire after the MVCC window, such false positives are limited to only happen within the short MVCC window time (i.e., 5 seconds).

The OCC design of FDB avoids the complicated logic of acquiring and releasing (logical) locks, which greatly simplifies interactions between the TS and the SS. The price is

wasted work done by aborted transactions. In our multi-tenant production workload transaction conflict rate is very low (less than 1%) and OCC works well. If a conflict happens, the client can simply restart the transaction.

2.2.3 Logging Protocol

After a `Proxy` decides to commit a transaction, it sends a message to all `LogServers`: mutations are sent to `LogServers` responsible for the modified key ranges, while other `LogServers` receive an empty message body. The log message header includes both the current and previous LSN obtained from the `Sequencer`, as well as the largest known committed version (KCV) of this `Proxy`. `LogServers` reply to the `Proxy` once the log data is made durable, and the `Proxy` updates its KCV to the LSN if all replica `LogServers` have replied and this LSN is larger than the current KCV.

Shipping the redo log from the LS to the SS is not a part of the commit path and is performed in the background. In FDB, `StorageServers` apply non-durable redo logs from `LogServers` to an in-memory index. In the common case, this happens before any read versions that reflect the commit are handed out to a client. Therefore, when client read requests reach `StorageServers`, the requested version (i.e., the latest committed data) is usually already available. If fresh data is not available to read at a `StorageServer` replica, the client either waits for the data to become available or reissues the request at another replica [18]. If both reads time out, the client can simply restart the transaction.

Since log data is already durable on `LogServers`, `StorageServers` can buffer updates in memory and persist batches of data to disks periodically, thus improving I/O efficiency.

2.2.4 Transaction System Recovery

Traditional database systems often employ the ARIES recovery protocol [30]. During recovery, the system processes redo log records from the last checkpoint by re-applying them to relevant data pages. This brings the database to a consistent state; transactions that were in-flight during the crash can be rolled back by executing the undo log records.

In FDB, recovery is purposely made very cheap—there is no need to apply undo log entries. This is possible because of a greatly simplifying design choice: redo log processing is the same as normal log forward path. In FDB, `StorageServers` pull logs from `LogServers` and apply them in the background. The recovery process starts by detecting a failure and recruits a new transaction system. The new TS can accept transactions before all the data in old `LogServers` is processed. Recovery only needs to find out the end of the redo log: At that point (as in normal forward operation) `StorageServers` asynchronously replay the log.

For each epoch, the `Sequencer` executes recovery in several steps. First, it reads the previous TS configuration from `Coordinators` and locks this information to prevent another `Sequencer` from recovering concurrently. Next, it recovers previous TS system states, including information about older `LogServers`, stops them from accepting transactions, and recruits a new set of `Proxies`, `Resolvers`, and `LogServers`. After previous `LogServers` are stopped and a new TS is recruited, the `Sequencer` writes the new TS information to the `Coordinators`. Because `Proxies` and `Resolvers` are stateless, their recoveries have no extra work. In contrast, `LogServers` save the logs of committed transactions, and we need to ensure all such transactions are durable and retrievable by `StorageServers`.

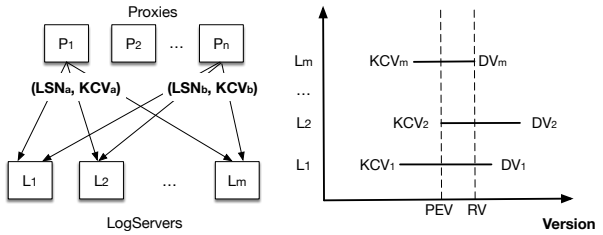


Figure 2: An illustration of RV and PEV.

The essence of the recovery of old `LogServers` is to determine the end of redo log, i.e., a Recovery Version (RV). Rolling back undo log is essentially discarding any data after RV in the old `LogServers` and `StorageServers`. Figure 2 illustrates how RV is determined by the `Sequencer`. Recall that a `Proxy` request to `LogServers` piggybacks its KCV, the maximum LSN that this `Proxy` has committed, along with the LSN of the current transaction. Each `LogServer` keeps the maximum KCV received and a Durable Version (DV), which is the maximum LSN persisted by the `LogServer` (DV is ahead of KCV since it corresponds to in-flight transactions). During a recovery, the `Sequencer` attempts to stop all m old `LogServers`, where each response contains the DV and KCV on that `LogServer`. Assume the replication degree for `LogServers` is k. Once the `Sequencer` has received more than $m - k$ replies, the `Sequencer` knows the previous epoch has committed transactions up to the maximum of all KCVs, which becomes the previous epoch's end version (PEV). All data before this version has been fully replicated. For current epoch, its start version is $PEV + 1$ and the `Sequencer` chooses the minimum of all DVs to be the RV. Logs in the range of $[PEV + 1, RV]$ are copied from previous epoch's `LogServers` to the current ones, for healing the replication degree in case of `LogServer` failures. The overhead of copying this range is very small because it only contains a few seconds' log data.

When `Sequencer` accepts new transactions, the first is a special recovery transaction that informs `StorageServers` the RV so that they can roll back any data larger than RV. The current FDB storage engine consists of an unversioned SQLite [24] B-tree and in-memory multi-versioned redo log data. Only mutations leaving the MVCC window (i.e., committed data) are written to SQLite. The rollback is simply discarding in-memory multi-versioned data in `StorageServers`. Then `StorageServers` pull any data larger than version PEV from new `LogServers`.

2.3 Replication

FDB uses a combination of various replication strategies for different data to tolerate f failures:

- *Metadata replication.* System metadata of the control plane is stored on `Coordinators` using Active Disk Paxos [15]. As long as a quorum (i.e., majority) of `Coordinators` are live, this metadata can be recovered.

- *Log replication.* When a `Proxy` writes logs to `LogServers`, each sharded log record is synchronously replicated on $k = f + 1$ `LogServers`. Only when all k have replied with successful persistence can the `Proxy` send back the commit response to the client. Failure of a `LogServer` results in a transaction system recovery.

- *Storage replication.* Every shard, i.e., a key range, is asynchronously replicated to $k = f + 1$ `StorageServers`, which is called a *team*. A `StorageServer` usually hosts a number of shards so that its data is evenly distributed across many teams. A failure of a `StorageServer` triggers `DataDistributor` to move data from teams containing the failed process to other healthy teams.

Note the storage team abstraction is more sophisticated than Copysets [17]. To reduce the chance of data loss due to simultaneous failures, FDB ensures that at most one process in a replica group is placed in a *fault domain*, e.g., a host, rack or availability zone. Each team is guaranteed to have at least one process live and there is no data loss if any one of the respective fault domains remains available.

3 Simulation Testing

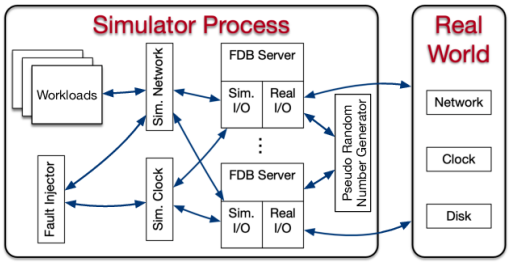


Figure 3: The FDB deterministic simulator.

Testing and debugging distributed systems is a challenging and inefficient process. This problem is particularly acute for FDB—any failure of its strong concurrency control contract can produce almost arbitrary corruption in systems layered on top. Accordingly, an ambitious approach to end-to-end testing was adopted from the beginning: the real database software is run, together with randomized synthetic workloads and fault injection, in a deterministic discrete-event simulation. The harsh simulated environment quickly provokes bugs in the database, and determinism guarantees that every such bug can be reproduced and investigated.

Deterministic simulator. FDB was built from the ground up to enable this testing approach. All database code is deterministic and multithreaded concurrency is avoided (instead, one database node is deployed per core). Figure 3 illustrates the simulator process of FDB, where all sources of nondeterminism and communication are abstracted, including network, disk, time, and pseudo random number generator. FDB is written in Flow [3], a novel syntactic extension to C++ adding async/await-like concurrency primitives. Flow provides the Actor programming model [10] that abstracts various actions of the FDB server process into a number of actors that are scheduled by the Flow runtime library. The simulator process is able to spawn multiple FDB servers that communicate with each other through a simulated network in a single discrete-event simulation.

The simulator runs multiple workloads (written in Flow) that communicate with simulated FDB servers through the simulated network. These workloads include fault injection instructions, mock applications, database configuration changes, and internal database functionality invocations.

Test oracles. FDB uses a variety of test oracles to detect failures in simulation. Most of the synthetic workloads have assertions built in to verify the contracts and properties of the database, e.g., by checking invariants in their data that can only be maintained through transaction atomicity and isolation. Assertions are used throughout the code-base to check properties that can be verified "locally". Properties like recoverability (eventual availability) can be checked by returning the modeled hardware environment (after a set of failures sufficient to break the database's availability) to a state in which recovery should be possible and verifying that the cluster eventually recovers.

Fault injection. Simulation injects machine, rack, and data-center failures and reboots, a variety of network faults, partitions, and latency problems, disk behavior (e.g. the corruption of unsynchronized writes when machines reboot), and randomizes event times. This variety of fault injection both tests the database's resilience to specific faults and increases the diversity of states in simulation. Fault injection distributions are carefully tuned to avoid driving the system into a small state-space caused by an excessive fault rate.

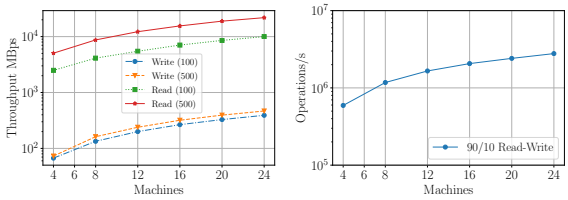
FDB itself cooperates with the simulation in making rare states and events more common, through a high-level fault injection technique informally referred to as "buggification". At many places in its code-base, the simulation is given the opportunity to inject some unusual (but not contract-breaking) behavior such as unnecessarily returning an error from an operation that usually succeeds, or choosing an unusual value for a tuning parameter, etc. This complements fault injection at the network and hardware level.

Swarm testing [23] is extensively used to maximize the diversity of simulation runs. Each run uses a random cluster size and configuration, random workloads, random fault injection parameters, random tuning parameters, and enables and disables a random subset of buggification points. We have open-sourced the swarm testing framework for FDB [6].

Latency to bug discovery. Finding bugs quickly is important both so that they are encountered in testing before production, and for engineering productivity (since bugs found immediately in an individual commit can be trivially traced to that commit). Discrete-event simulation can run arbitrarily faster than real-time if CPU utilization within the simulation is low, as the simulator can fast-forward clock to the next event. Many distributed systems bugs take time to play out, and running simulations with long stretches of low utilization allows many more of these to be found per core second than in "real-world" end-to-end tests.

Additionally, randomized testing is embarrassingly parallel and FDB developers can and do "burst" the amount of testing they do before major releases. Since the search space is effectively infinite, running more tests results in more code being covered and more potential bugs being found.

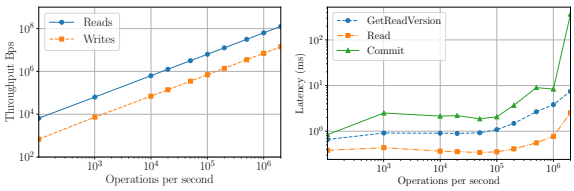
Limitations. Simulation cannot reliably detect performance issues, such as an imperfect load balancing algorithm. It is also unable to test third-party libraries or dependencies, or even first-party code not implemented in Flow. As a consequence, we have largely avoided taking dependencies on external systems. Finally, bugs in critical dependent systems, such as a filesystem or the operating system, or misunderstandings of their contract, can lead to bugs in FDB. For example, several bugs have resulted from the true operating system contract being weaker than it was believed to be.



(a) Read and write throughput with 100 and 500 operations per transaction.

(b) 90/10 Read-write.

Figure 4: Scalability test.



(a) Throughput.

(b) Average latency.

Figure 5: Throughput and average latency for different operation rate on a 24-machine cluster configuration.

4 Evaluation

This section studies the scalability of FDB and provides some data on the time of reconfiguration.

4.1 Scalability Test

The experiments were conducted on a test cluster of 27 machines in a single data center. Each machine has a 16-core 2.5 GHz Intel Xeon CPU with hyper-threading enabled, 256 GB memory, 8 SSD disks, connected via 10 Gigabit Ethernet. Each machine runs 14 `StorageServers` on 7 SSD disks and reserves the remaining SSD for `LogServer`. In the experiments, we use the same number of `Proxies` and `LogServers`. The replication degrees for both `LogServers` and `StorageServers` are set to three.

We use a synthetic workload to evaluate the performance of FDB. Specifically, there are four types of transactions: 1) *blind writes* that update a configured number of random keys; 2) *range reads* that fetch a configured number of continuous keys starting at a random key; 3) *point reads* that fetch 10 random keys; and 4) *point writes* that fetch 5 random keys and update another 5 random keys. We use blind writes and range reads to evaluate the write and read performance, respectively. Point reads and point writes are used together to evaluate mixed read-write performance. For instance, 90% reads and 10% writes (90/10 read-write) is constructed with 80% point reads and 20% point writes transactions. Each key is 16 bytes and the value size is uniformly distributed between 8 and 100 bytes (averaging 54 bytes). The database is pre-populated with data using the same size distribution. In the experiments, we make sure the dataset cannot be completely cached in the memory of `StorageServers`.

Figure 4 illustrates the scalability test of FDB from 4 to 24 machines using 2 to 22 `Proxies` or `LogServers`. Figure 4a shows that the write throughput scales from 67 to 391 MBps (5.84X) for 100 operations per transaction (T100), and from 73 to 467 MBps (6.40X) for 500 operations per transaction (T500). Note the raw write throughput is three times higher, because each write is replicated three times to `LogServers` and `StorageServers`. `LogServers` are CPU saturated at the maximum write throughput. Read throughput increases from 2,946 to 10,096 MBps (3.43X) for T100, and from 5055 to 21,830 MBps (4.32X) for T500, where `StorageServers` are saturated. For both reads and writes, increasing the number operations in a transaction boosts throughput. However, increasing operations further (e.g. to 1000) doesn't bring significant changes. Figure 4b shows the operations per second for 90/10 read-write traffic, which increases from 593k to 2,779k (4.69X). In this case, `Resolvers` and `Proxies` are CPU saturated.

The above experiments study saturated performance. Figure 5 illustrates the client performance on a 24-machine cluster with varying operation rate of 90/10 read-write load. This configuration has 2 `Resolvers`, 22 `LogServers`, 22 `Proxies`, and 336 `StorageServers`. Figure 5a shows that the throughput scales linearly with more operations per second (Ops) for both reads and writes. For latency, Figure 5b shows that when Ops is below 100k, the mean latencies remain stable: about 0.35ms to read a key, 2ms to commit, and 1ms to get a read version (GRV). Read is a single hop operation, thus is faster than the two-hop GRV request. The commit request involves multiple hops and persistence to three `LogServers`, thus higher latency than reads and GRVs. When Ops exceeds 100k, all these latencies increase because of more queuing time. At 2m Ops, `Resolvers` and `Proxies` are saturated. Batching helps to sustain the throughput, but commit latency spike to 368ms due to saturation.

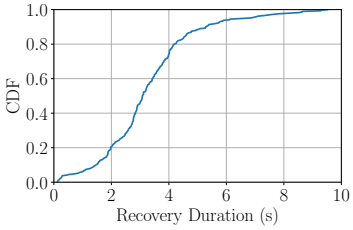


Figure 6: CDF plot for reconfiguration duration in seconds.

4.2 Reconfiguration Duration

We collected 289 reconfiguration (i.e., transaction system recovery) traces from our production clusters that typically host hundreds of TBs data. Because of the client-facing nature, short reconfiguration time is critical for the high availability of these clusters. Figure 6 illustrates the cumulative distribution function (CDF) of the reconfiguration times. The median and 90-percentile are 3.08 and 5.28 seconds, respectively. The reason for these short recovery times is that they are not bounded by the data or transaction log size, and are only related to the system metadata sizes. During the recovery, read-write transactions were temporarily blocked and were retried after timeout. However, client reads were not impacted. The causes of these reconfigurations include automatic failure recovery from software or hardware faults, software upgrades, database configuration changes, and the manual mitigation of production issues.

5 Lessons Learned

This section discusses our experience and lessons of FDB.

Architecture Design. The divide-and-conquer design principle has proven to be an enabling force for flexible cloud deployment, making the database extensible as well as performant. First, separating the transaction system from the storage layer enables greater flexibility in placing and scaling compute and storage resources independently. An added benefit of `LogServers` is that they are akin to witness replicas; in some of our multi-region production deployments, `LogServers` significantly reduce the number of `StorageServers` (full replicas) required to achieve the same high-availability properties. Further, operators are free to place heterogeneous roles of FDB on different server instance types, optimizing for performance and costs. Second, the decoupling design makes it possible to extend the database functionality, such as our ongoing work of supporting RocksDB [21] as a drop-in replacement for the current SQLite engine. Finally, many of the recent performance improvements are specializing functionality as dedicated roles, e.g., separating `DataDistributor` and `Ratekeeper` from `Sequencer`, adding storage cache, dividing `Proxies` into get-read-version proxy and commit proxy. This design pattern successfully allows new features and capabilities to be added frequently.

Simulation Testing. Simulation testing has enabled FDB to maintain a very high development velocity with a small team by shortening the latency between a bug being introduced and a bug being found, and by allowing deterministic reproduction of issues. Adding additional logging, for instance, generally does not affect the deterministic ordering of events, so an exact reproduction is guaranteed. The productivity of this debugging approach is so much higher than normal production debugging, that in the rare circumstances when a bug was first found "in the wild", the debugging process was almost always first to improve the capabilities or the fidelity of the simulation until the issue could be reproduced there, and only then to begin the normal debugging process. Rigorous correctness testing via simulation makes FDB extremely reliable. In the past several years, CloudKit [32] has deployed FDB for more than 0.5M disk years without a single data corruption event.

It is hard to measure the productivity improvements stemming from increased confidence in the testability of the system. On numerous occasions, the FDB team executed ambitious, ground-up rewrites of major subsystems. Without simulation testing, many of these projects would have been deemed too risky or too difficult, and not even attempted.

The success of simulation has led us to continuously push the boundary of what is amenable to simulation testing by eliminating dependencies and reimplementing them ourselves in Flow. For example, early versions of FDB depended on Apache Zookeeper for coordination, which was deleted after real-world fault injection found two independent bugs in Zookeeper (circa 2010) and was replaced by a de novo Paxos implementation written in Flow. No production bugs have ever been reported since.

Fast Recovery. Fast recovery is not only useful for improving availability, but also greatly simplifies the software upgrades and configuration changes and makes them faster. Traditional wisdom of upgrading a distributed system is to perform rolling upgrades so that rollback is possible when something goes wrong. The duration of rolling upgrades can last from hours to days. In contrast, FoundationDB upgrades can be performed by restarting all processes at the same time, which usually finishes within a few seconds. Because this upgrade path has been extensively tested in simulation, all upgrades in Apple's production clusters are performed in this way. Additionally, this upgrade path simplifies protocol compatibility between different versions—we only need to make sure on-disk data is compatible. There is no need to ensure the compatibility of RPC protocols between different software versions.

5s MVCC Window. FDB chooses a 5-second MVCC window to limit the memory usage of the transaction system and storage servers, because the multi-version data is stored in the memory of `Resolvers` and `StorageServers`, which in turn restricts transaction sizes. From our experience, this 5s window is long enough for the majority of OLTP use cases. If a transaction exceeds the time limit, it is often the case that the client application is doing something inefficient, e.g., issuing reads one by one instead of parallel reads. As a result, exceeding the time limit often exposes inefficiency in the application.

For some transactions that may span more than 5s, many can be divided into smaller transactions. For instance, the continuous backup process of FDB will scan through the key space and create snapshots of key ranges. Because of the 5s limit, the scanning process is divided into a number of smaller ranges so that each range can be performed within 5s. In fact, this is a common pattern: one transaction creates a number of jobs and each job can be further divided or executed in a transaction. FDB has implemented such a pattern in an abstraction called `TaskBucket` and the backup system heavily depends on it.

6 Related Work

NoSQL systems such as BigTable [14], Dynamo [19], MongoDB [9], Cassandra [26] do not provide ACID transactions. Google's Percolator [31]and Omid [13] layered transactional APIs atop key value stores with snapshot isolation. FDB supports strict serializable ACID transactions on a scalable key-value store that has been used to support flexible schema and richer queries [5, 16, 28].

Traditional bundled database systems have tight coupling of the transaction component and data component [20, 33]. Unbundled database systems separate transaction component (TC) from data component (DC) [29], and typically adopt lock-based concurrency control. In FDB, TC is further decomposed into a number of dedicated roles and the transaction logging is decoupled from TC. As a result, FDB chooses OCC with a deterministic transaction order.

Non-deterministic fault-injection has been widely used in the testing of distributed systems, such as network partition [11], power failures [36], storage faults [22], and Jepsen testing [2]. All of these approaches lack deterministic reproducibility. While model checking [27, 35] can be more exhaustive than simulation, it can only verify the correctness of a model rather than of the actual implementation. The FDB deterministic simulation approach allows verification of database invariants and other properties against the real database code, together with deterministic reproducibility.

7 Conclusions

FoundationDB is a key value store designed for OLTP cloud services. The main idea is to decouple transaction processing from logging and storage. Such an unbundled architec-

ture enables the separation and horizontal scaling of both read and write handling. The transaction system combines OCC and MVCC to ensure strict serializability. The decoupling of logging and the determinism in transaction orders greatly simplify recovery, thus allowing unusually quick recovery time and improving availability. Finally, deterministic and randomized simulation has ensured the correctness of the database implementation.

8 References

[1] CouchDB. `https://couchdb.apache.org/`.

[2] Distributed system safety research. `https://jepsen.io/`.

[3] Flow. `https://github.com/apple/foundationdb/tree/master/flow`.

[4] FoundationDB. `https://github.com/apple/foundationdb`.

[5] FoundationDB Document Layer. `https://github.com/FoundationDB/fdb-document-layer`.

[6] FoundationDB Joshua. `https://github.com/FoundationDB/fdb-joshua`.

[7] Foundationdb storage adapter for janusgraph. `https://github.com/JanusGraph/janusgraph-foundationdb`.

[8] Janusgraph. `https://janusgraph.org/`.

[9] MongoDB. `https://www.mongodb.com/`.

[10] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[11] A. Alquraan, H. Takruri, M. Alfatafta, and S. Al-Kiswany. An analysis of network-partitioning failures in cloud systems. In *USENIX OSDI*, 2018.

[12] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[13] E. Bortnikov, E. Hillel, I. Keidar, I. Kelly, M. Morel, S. Paranjpye, F. Perez-Sorrosal, and O. Shacham. Omid, reloaded: Scalable and highly-available transaction processing. In *USENIX FAST*, 2017.

[14] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *USENIX OSDI*, 2006.

[15] G. Chockler and D. Malkhi. Active disk paxos with infinitely many processes. In *ACM PODC*, 2002.

[16] C. Chrysafis, B. Collins, S. Dugas, J. Dunkelberger, M. Ehsan, S. Gray, A. Grieser, O. Herrnstadt, K. Lev-Ari, T. Lin, M. McMahon, N. Schiefer, and A. Shraer. FoundationDB Record Layer: A Multi-Tenant Structured Datastore. In *ACM SIGMOD*, 2019.

[17] A. Cidon, S. Rumble, R. Stutsman, S. Katti, J. Ousterhout, and M. Rosenblum. Copysets: Reducing the frequency of data loss in cloud storage. In *USENIX Annual Technical Conference*, 2013.

[18] J. Dean and L. A. Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, Feb. 2013.

[19] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. *ACM SIGOPS Operating Systems Review*, 41(6):205–220, 2007.

[20] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: Sql server's memory-optimized oltp engine. In *ACM SIGMOD*, 2013.

[21] Facebook. Rocksdb. `https://rocksdb.org`.

[22] A. Ganesan, R. Alagappan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Redundancy does not imply fault tolerance: Analysis of distributed storage reactions to single errors and corruptions. In *USENIX FAST*, 2017.

[23] A. Groce, C. Zhang, E. Eide, Y. Chen, and J. Regehr. Swarm testing. In *ACM ISSTA*, 2012.

[24] R. D. Hipp. SQLite. `https://www.sqlite.org/index.html`, 2020.

[25] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, June 1981.

[26] A. Lakshman and P. Malik. Cassandra: structured storage system on a p2p network. In *ACM PODC*, 2009.

[27] T. Leesatapornwongsa, M. Hao, P. Joshi, J. F. Lukman, and H. S. Gunawi. Samc: Semantic-aware model checking for fast discovery of deep bugs in cloud systems. In *USENIX OSDI*, 2014.

[28] K. Lev-Ari, Y. Tian, A. Shraer, C. Douglas, H. Fu, A. Andreev, K. Beranek, S. Dugas, A. Grieser, and J. Hemmo. Quick: a queuing system in cloudkit. In *ACM SIGMOD*, 2021.

[29] D. Lomet, A. Fekete, G. Weikum, and M. J. Zwilling. Unbundling transaction services in the cloud. In *CIDR*, 2009.

[30] C. Mohan, D. Haderle, B. G. Lindsay, H. Pirahesh, and P. M. Schwarz. Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems*, 1992.

[31] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *USENIX OSDI*, 2010.

[32] A. Shraer, A. Aybes, B. Davis, C. Chrysafis, D. Browning, E. Krugler, E. Stone, H. Chandler, J. Farkas, J. Quinn, J. Ruben, M. Ford, M. McMahon, N. Williams, N. Favre-Felix, N. Sharma, O. Herrnstadt, P. Seligman, R. Pisolkar, S. Dugas, S. Gray, S. Lu, S. Harkema, V. Kravtsov, V. Hong, Y. Tian, and W. L. Yih. Cloudkit: Structured storage for mobile applications. *Proceedings of the VLDB Endowment*, 11(5), Jan. 2018.

[33] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *ACM SOSP*, 2013.

[34] M. Yabandeh and D. Gómez Ferro. A Critique of Snapshot Isolation. In *ACM EuroSys*, 2012.

[35] J. Yang, T. Chen, M. Wu, Z. Xu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang, and L. Zhou. MODIST: transparent model checking of unmodified distributed systems. In *USENIX NSDI*, 2009.

[36] M. Zheng, J. Tucek, D. Huang, F. Qin, M. Lillibridge, E. S. Yang, B. W. Zhao, and S. Singh. Torturing databases for fun and profit. In *USENIX OSDI*, 2014.

Technical Perspective of TURL: Table Understanding through Representation Learning

Paolo Papotti
papotti@eurecom.fr
EURECOM

Several efforts aim at representing tabular data with neural models for supporting target applications at the intersection of natural language processing (NLP) and databases (DB) [1–3]. The goal is to extend to structured data the recent neural architectures, which achieve state of the art results in NLP applications. Language models (LMs) are usually pre-trained with unsupervised tasks on a large text corpus. The output LM is then fine-tuned on a variety of downstream tasks with a small set of specific examples. This process has many advantages, because the LM contains information about textual structure and content, which are used by the target application without manually defining features.

Language models that consume tables and text can enable solutions that go beyond the limits of traditional declarative specifications built around SQL, such as answering queries expressed in natural language [3], computational fact-checking of textual claims [2], data integration [1], and text generation. Indeed, there is consensus across the two communities of the importance of developing models for representing and "understanding" tables accurately in applications that involve both structured data and natural language. Given the success of transformers in developing pre-trained LMs, such as BERT, there are several proposals to enhance their architecture for representing relational tables.

In this landscape, TURL stands as one of the reference approaches with three main contributions. First, it is among the first solutions extending the transformer architecture to consume structured data and text during pre-training. This is crucial, as it allows the generation of contextual embeddings, which naturally outperform in applications the previous methods based on static embeddings, e.g., tuples linearized and encoded with Word2Vec [1]. Second, it shows how to effectively model the structure in relational tables, with clear notions of tuples and columns. Information expressed in tables cannot be modeled without explicitly capturing such relationships across cell values. Third, it shows the benefits of the fine tuning approach with promising qualitative results over six traditional DB tasks. While the benefits for NLP tasks are well documented, the results over these DB target applications are especially important to show the potential of the LMs with structured data.

To achieve these results, the technical advancements in TURL span across the transformer architecture. To properly model the structure of data in tables, the original transformer is extended and updated by modifying components at different levels. At the *embedding level*, modifications include additional embeddings to explicitly model the table structure and differentiate entity types. At the *encoder level*, a masked self-attention module attends to structurally related elements, such as elements in a row or a column, unlike the traditional transformer where each element attends to all other elements in the sequence. Finally, in terms of *training*, it introduces a reconstruction task with the Masked Entity Recovery pre-training objective. The idea is to reconstruct the correct input from a corrupted one, i.e., randomly mask out entities in the table with the objective of recovering them based on the remaining entities and the sentences associated to the table, such as captions.

Stepping back to look at the bigger picture, the methods proposed in this paper enable the generation of data structure-aware LMs, which can be a fundamental tool to push progress in target applications using structured data. From this perspective, it is clear that TURL makes an important contribution to the general quest of bridging the gap between the NLP and the DB communities.

REFERENCES

[1] Riccardo Cappuzzo, Paolo Papotti, and Saravanan Thirumuruganathan. 2020. Creating Embeddings of Heterogeneous Relational Datasets for Data Integration Tasks. In *SIGMOD*.

[2] Rami Aly et al. 2021. FEVEROUS: Fact Extraction and VERification Over Unstructured and Structured information. In *NeurIPS - Datasets and Benchmarks Track*.

[3] Jonathan Herzig, Pawel Krzysztof Nowak, Thomas Mueller, Francesco Piccinno, and Julian Eisenschlos. 2020. TaPas: Weakly Supervised Table Parsing via Pre-training. In *ACL*. 4320–4333.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

TURL: Table Understanding through Representation Learning

Xiang Deng*
The Ohio State University
Columbus, OH
deng.595@osu.edu

Huan Sun*
The Ohio State University
Columbus, OH
sun.397@osu.edu

Alyssa Lees
Google Research
New York, NY
alyssalees@google.com

You Wu
Google Research
New York, NY
wuyou@google.com

Cong Yu
Google Research
New York, NY
congyu@google.com

ABSTRACT

Relational tables on the Web store a vast amount of knowledge. Owing to the wealth of such tables, there has been tremendous progress on a variety of tasks in the area of table understanding. However, existing work generally relies on heavily-engineered task-specific features and model architectures. In this paper, we present TURL, a novel framework that introduces the pre-training/fine-tuning paradigm to relational Web tables. During pre-training, our framework learns deep contextualized representations on relational tables in a self-supervised manner. Its universal model design with pre-trained representations can be applied to a wide range of tasks with minimal task-specific fine-tuning.

Specifically, we propose a structure-aware Transformer encoder to model the row-column structure, and present a new Masked Entity Recovery (MER) objective for pre-training to capture relational knowledge. We compiled a benchmark consisting of 6 different tasks for table understanding and used it to systematically evaluate TURL. We show that TURL generalizes well to all tasks and substantially outperforms existing methods in almost all instances.

1. INTRODUCTION

Relational tables are in abundance on the Web and store a large amount of knowledge. Each relational Web table has its own "schema" of labeled and typed columns and essentially forms a small structured database [3]. Over the past decade, various large-scale collections of such tables have been aggregated [3, 4, 2, 21]. Owing to the wealth and utility of these tables, various tasks such as table interpretation [2, 35, 26, 36, 14] and table augmentation [8, 33, 1, 34, 3] have made tremendous progress in the past few years.

However, previous work [33, 34, 2] often relies on heavily-engineered task-specific methods such as using simple statistical/language features or straightforward string matching. These techniques suffer from several disadvantages. First, simple features only capture shallow patterns and often fail to handle the flexible schema and varied expressions in relational tables. Second, task-specific features and model architectures require a lot of effort to design and do not generalize well across tasks.

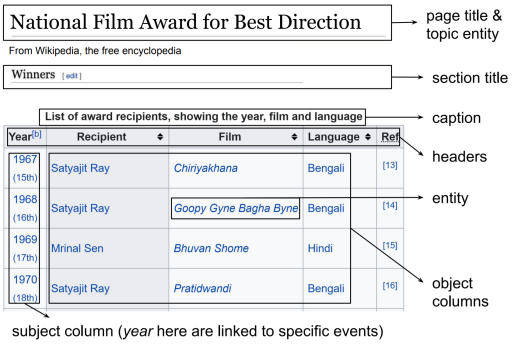


Figure 1: An example of a relational table from Wikipedia.

On the other hand, the representation learning model of [9], which uses Word2Vec [24] to learn general-purpose embedding vectors for words and entities in tables, achieved promising results on some table based tasks. However, [9] cannot generate *contextualized* representations, i.e., it does not consider varied use of words/entities in different contexts and only produces a single fixed embedding vector for word/entity. In addition, *shallow* neural models like Word2Vec have relatively limited learning capabilities, which hinder the capture of complex semantic knowledge contained in relational tables.

We propose TURL, a novel framework for learning deep contextualized representations on relational tables via self-supervised pre-training and task-specific fine-tuning. Recently, such pre-training and fine-tuning paradigms have achieved remarkable success on unstructured text data [13]. In contrast, little effort has been extended to study them on relational tables. Our work fills this gap.

There are two main challenges in the development of TURL: (1) *Relational table encoding.* Existing neural network encoders are designed for linearized sequence input and are a good fit with unstructured texts. However, data in relational tables is organized in a semi-structured format. Moreover, a relational table contains multiple components including table caption, headers and cell values. The challenge is to develop a means of modeling the row-and-column structure

*Corresponding authors.

DOI: 10.14778/3430915.3430921.

as well as integrating the information from different components of the table. (2) *Factual knowledge modeling.* Pre-trained language models such as BERT [13] focus on modeling the syntactic and semantic characteristics of word use in natural sentences. However, relational tables contain a vast amount of factual knowledge about entities, which cannot be captured by existing language models directly. Effectively modelling such knowledge in TURL is a second challenge.

To address the first challenge, we encode information from different table components into separate input embeddings and fuse them together. We then employ a *structure-aware* Transformer [28] encoder with masked self-attention. We explicitly model the row-and-column structure by restraining each element to only aggregate information from other structurally related elements. To achieve this, we build a visibility matrix based on the table structure and use it as an additional mask for the self-attention layer [28]. For the second challenge, we learn embeddings for each entity during pre-training, and model the relation between entities with the assistance of the visibility matrix. We then propose a Masked Entity Recovery (MER) pre-training objective: We randomly mask out entities in a table and predict the masked entities based on other entities and the table context (e.g., caption/header). *This encourages the model to learn factual knowledge from the tables and encode it into entity embeddings.* In addition, for a certain percentage of masked entities, we utilize their entity mention as additional information to base the recovery on. *This helps our model build connections between words and entities.*

We pre-train our model on around 570K relational tables from Wikipedia and fine-tune it for specific downstream tasks. A distinguishing feature of TURL is its universal architecture across different tasks - only minimal modification is needed to cope with each downstream task. To facilitate research in this direction, we compiled a benchmark that consists of 6 table understanding tasks, including entity linking, column type annotation, relation extraction, row population, cell filling and schema augmentation. Experimental results show that TURL substantially outperforms existing task-specific and shallow representation learning methods [7, 14, 26, 17, 33, 9].

This paper is a shortened version of our paper with the same title that appeared in VLDB 2021 [11]. More technical details and experimental results can be found in [11], as well as in the extended version published on Arxiv [12]. Our source code, benchmark, as well as pre-trained models are available online to facilitate future research.[1]

2. PRELIMINARY

In this work, we focus on relational Web tables and are most interested in the factual knowledge about entities. Each table T is associated with the following: (1) Table caption C, which is a short text description summarizing what the table is about. (2) Table headers H, which define the table schema; (3) Topic entity e_t, which describes what the table is about and is usually extracted from the table caption or page title; (4) Table cells E containing entities. Each entity cell $e \in E$ contains a specific object with a unique identifier. For each cell, we define the entity as $e = (e^{\mathrm{e}}, e^{\mathrm{m}})$, where e^{e} is the specific entity linked to the cell and e^{m} is the entity mention (i.e., the text string).

[1] https://github.com/sunlab-osu/TURL

(C, H, e_t) is also known as *table metadata* and E is the actual *table content*. We study self-supervised representation learning on relational Web tables, defined as follows.

Definition. Given a relational Web table corpus, we aim to learn in a self-supervised manner a task-agnostic contextualized vector representation for each token in all table captions C's and headers H's and for each entity (i.e., all entity cells E's and topic entities e_t's).

3. RELATED WORK

Representation Learning. The pre-training/fine-tuning paradigm has drawn tremendous attention in recent years. Extensive effort has been devoted to the development of self-supervised representation learning methods for both unstructured text [24, 13] and structured knowledge bases (KB) [27], which in turn can be utilized for a wide variety of downstream tasks via fine-tuning.

Despite the success of representation learning on text and KB, few works have thoroughly explored representation learning on tables. Pre-trained language models are directly adopted in [22] for entity matching. Two recent papers from the NLP community [16, 31] study pre-training on Web tables to assist in semantic parsing or question answering tasks on tables. In this work, we introduce TURL, a new methodology for learning deep contextualized representations for relational Web tables that preserve both semantic and knowledge information. In addition, we conduct comprehensive experiments on a much wider range of table-related tasks.

Table Understanding. In general, there are two types of table understanding tasks: table interpretation and table augmentation. Table interpretation aims to uncover the semantic attributes of the data contained in relational tables, and transform this information into machine understandable knowledge. This task is usually accomplished with help from existing KBs. In turn, the extracted knowledge can be used for KB construction and population. There are three main tasks for table interpretation: entity linking [2, 26, 14], column type annotation [6, 17] and relation extraction [26, 36]. Table augmentation seeks to offer intelligent assistance to the user when composing tables by expanding a seed query table with additional data. Specifically, for relational tables this can be divided into three sub-tasks: row population [8, 33], cell filling [1, 34] and schema augmentation [3, 33].

Several benchmarks have been proposed for both table interpretation [21, 23, 14, 19] and table augmentation [33, 34] over the last decade. Although these benchmarks have been used in various recent studies, they still suffer from a few shortcomings: (1) They are typically small sets of sampled tables with limited annotations. (2) SemTab 2019 [19] contains a large number of instances; however, most of them are automatically generated and lack metadata/context of the Web tables. In this work, we compile a larger benchmark covering both table interpretation and table augmentation tasks. We also use some of these existing datasets for more comprehensive evaluation. By leveraging large-scale relational tables on Wikipedia and a curated KB, we ensure both the size and quality of our dataset.

4. METHODOLOGY

In this section, we introduce our TURL framework for self-supervised representation learning on relational tables. Figure 3 presents an overview of TURL which consists of

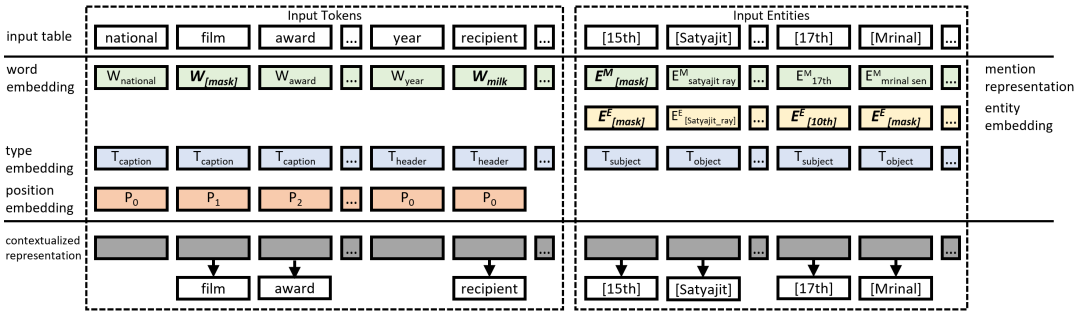


Figure 2: Illustration of the model input-output. The input table is first transformed into a sequence of tokens and entity cells, and masked for pre-training as described in Section 4.3. We then get contextualized representations for the table and use them for pre-training. Here [15th] (which means 15th National Film Awards), [Satyajit], ... are linked entity cells.

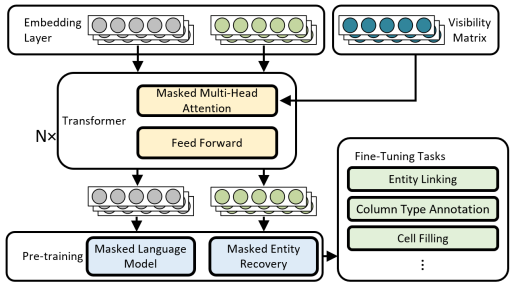


Figure 3: Overview of our TURL framework.

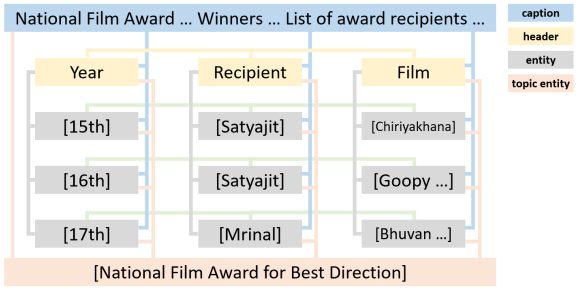


Figure 4: Graphical illustration of masked self-attention by our visibility matrix. Each token/entity in a table can only attend to its directly connected neighbors (shown as edges here, $M_{i,j} = 1$ only if two elements are connected).

three modules: (1) An embedding layer to convert different components of an input table into input embeddings, (2) N stacked structure-aware Transformer [28] encoders to capture the textual information and relational knowledge, and (3) a final projection layer for pre-training objectives. Figure 2 shows an input-output example.

4.1 Embedding Layer

Given a table $T=(C, H, E, e_t)$, we first linearize the input into a sequence of tokens and entity cells by concatenating the metadata and scanning the content row by row. The embedding layer then converts each token in C and H and each entity in E and e_t into an embedding representation.

Input token representation. For each token w, its vector representation is obtained as follows:

$$\mathbf{x}^{\mathrm{t}} = \mathbf{w} + \mathbf{t} + \mathbf{p}. \quad (1)$$

Here $\mathbf{w}$ is the word embedding vector, $\mathbf{t}$ is called the type embedding vector and aims to differentiate whether token w is in the table caption or a header, and $\mathbf{p}$ is the position embedding vector that provides relative position information for a token within the caption or a header.

Input entity representation. For each entity cell $e = (e^{\mathrm{e}}, e^{\mathrm{m}})$ (same for topic entity e_t), we fuse the information from the linked entity e^{e} and entity mention e^{m} together, and use an additional type embedding vector $\mathbf{t}^{\mathrm{e}}$ to differentiate three types of entity cells (i.e., subject/object/topic). Specifically, we calculate the input entity representation $\mathbf{x}^{\mathrm{e}}$ as:

$$\mathbf{x}^{\mathrm{e}} = \texttt{LINEAR}([\mathbf{e}^{\mathrm{e}}; \mathbf{e}^{\mathrm{m}}]) + \mathbf{t}^{\mathrm{e}}; \quad (2)$$

$$\mathbf{e}^{\mathrm{m}} = \texttt{MEAN}(\mathbf{w}_1, \mathbf{w}_2, \ldots, \mathbf{w}_j, \ldots). \quad (3)$$

Here $\mathbf{e}^{\mathrm{e}}$ is the entity embedding learned during pre-training. To represent entity mention e^{m}, we use its average word embedding $\mathbf{w}_j$'s. `LINEAR` is a linear layer to fuse $\mathbf{e}^{\mathrm{e}}$ and $\mathbf{e}^{\mathrm{m}}$.

A sequence of token and entity representations ($\mathbf{x}^{\mathrm{t}}$'s and $\mathbf{x}^{\mathrm{e}}$'s) are then fed into the structure-aware Transformer encoder to produce contextualized representations.

4.2 Structure-aware Transformer Encoder

We choose Transformer [28] as our base encoder block, which is composed of a multi-head self-attention layer followed by a fully connected layer. Transformer has been widely used in pre-trained language models [13, 25]. However, Transformer is originally designed for unstructured text sequences and cannot model the row-column structure, which is important for interpreting relational tables. We propose a visibility matrix M to model such structure information in a relational table.

Our visibility matrix acts as an attention mask so that each token (or entity) can only aggregate information from other structurally related tokens/entities during the self-attention calculation [28]. M is a symmetric binary matrix with $M_{i,j} = 1$ if and only if $\texttt{element}_j$ is visible to $\texttt{element}_i$. The `element` here can be a token in the caption or header, or an entity in a table cell. Specifically, we define M as follows:

- If $\texttt{element}_i$ is the topic entity or a token in table caption, $\forall j, M_{i,j} = 1$. *Table caption and topic entity are visible to all components of the table.*
- If $\texttt{element}_i$ is a token or an entity in the table and $\texttt{element}_j$ is a token or an entity in the same row or column, $M_{i,j} = 1$. *Entities and text content in the same row or the same column are visible to each other.*

4.3 Pre-training Objective

In order to pre-train our model on an unlabeled table corpus, we adopt the Masked Language Model (MLM) objec-

tive from BERT to learn representations for tokens in table metadata and propose a Masked Entity Recovery (MER) objective to learn entity cell representations.

Masked Language Model. We adopt the same MLM objective as BERT, which trains the model to capture the lexical, semantic and contextual information described by table metadata. Given an input token sequence including table caption and table headers, we simply mask some percentage of the tokens at random, and then predict these masked tokens. We adopt the same percentage settings as BERT. The data processor selects 20% of the token positions at random. For a selected position, (1) 80% of the time we replace it with a special [MASK] token, (2) 10% of the time we replace it with another random token, and (3) 10% of the time we keep it unchanged.

Given a token position selected for MLM with representation $\mathbf{h}^{\mathrm{t}}$ output by our encoder, the probability of predicting its original token $w \in \mathcal{W}$ is then calculated as:

$$P(w) = \frac{\exp\left(\texttt{LINEAR}(\mathbf{h}^{\mathrm{t}}) \cdot \mathbf{w}\right)}{\sum_{w_k \in \mathcal{W}} \exp\left(\texttt{LINEAR}(\mathbf{h}^{\mathrm{t}}) \cdot \mathbf{w}_k\right)}. \quad (4)$$

Masked Entity Recovery. We propose a novel Masked Entity Recovery (MER) objective to help the model capture the factual knowledge embedded in the table content as well as the associations between table metadata and table content. Essentially, we mask a certain percentage of input entity cells and then recover the linked entity based on surrounding entity cells and table metadata. This requires the model to infer the relation between entities from table metadata and to encode the knowledge in entity embeddings.

We also take advantage of entity mentions. Specifically, as shown in Eqn. 2, the input entity representation has two parts: the embedding $\mathbf{e}^{\mathrm{e}}$ and the mention representation $\mathbf{e}^{\mathrm{m}}$. For some percentage of masked entity cells, we only mask $\mathbf{e}^{\mathrm{e}}$, and as such the model receives additional entity mention information to help form predictions. This assists the model in building a connection between entity embeddings and entity mentions, and helps downstream tasks where only cell texts are available.

Specifically, the data processor first chooses 60% of entity cells at random. Here we adopt a higher masking ratio for MER compared with MLM, because oftentimes in downstream tasks, none or few entities are given. For one chosen entity cell, (1) 10% of the time we keep both $\mathbf{e}^{\mathrm{m}}$ and $\mathbf{e}^{\mathrm{e}}$ unchanged, (2) 63% (i.e., 70% of the left 90%) of the time we mask both $\mathbf{e}^{\mathrm{m}}$ and $\mathbf{e}^{\mathrm{e}}$, (3) 27% of the time we keep $\mathbf{e}^{\mathrm{m}}$ unchanged and mask $\mathbf{e}^{\mathrm{e}}$ (among which we replace $\mathbf{e}^{\mathrm{e}}$ with embedding of a random entity to inject noise in 10% of the time). In both MLM and MER, we keep a certain portion of the selected positions unchanged so that the model can generate good representations for non-masked tokens/cells.

Given an entity cell selected for MER with a contexutalized representation $\mathbf{h}^{\mathrm{e}}$ output by our encoder, the probability of predicting entity $e \in \mathcal{E}$ is then calculated as follows:

$$P(e) = \frac{\exp\left(\texttt{LINEAR}(\mathbf{h}^{\mathrm{e}}) \cdot \mathbf{e}^{\mathrm{e}}\right)}{\sum_{e_k \in \mathcal{E}} \exp\left(\texttt{LINEAR}(\mathbf{h}^{\mathrm{e}}) \cdot \mathbf{e}^{\mathrm{e}}_k\right)}. \quad (5)$$

In reality, considering the entity vocabulary $\mathcal{E}$ is quite large, we only use the above equation to rank entities from a given candidate set. For efficient training, we construct the candidate set with (1) entities in the current table, (2) entities that have co-occurred with those in the current table, and (3) randomly sampled negative entities.

We use cross-entropy loss for both MLM and MER objectives and the final pre-training loss is given as follows:

$$loss = \sum \log\left(P(w)\right) + \sum \log\left(P(e)\right), \quad (6)$$

where the sums are over all tokens and entity cells selected in MLM and MER respectively.

Pre-training details. In this work, we leverage a pre-trained TinyBERT [18] model to initialize our structure-aware Transformer encoder. We use the Adam [20] optimizer with a linearly decreasing learning rate. The initial learning rate is 1e-4 and we pre-train the model for 80 epochs.

4.4 Pre-training Dataset Construction

We construct the pre-training data based on the WikiTable corpus [2] which contains around 1.65M tables extracted from Wikipedia pages. We process and extract relational tables from the corpus, which are further partitioned into the pre-training and held-out validation/test sets.

Pre-processing. For each table, we concatenate the page title and section title with the table caption to obtain a more comprehensive description. For each cell, we leverage hyperlinks to Wikipedia pages in it to normalize different entity mentions corresponding to the same entity. To identify relational tables, we first locate entity columns that contain at least one linked cell and have meaningful headers (i.e., not *note*, digit numbers, etc.). We then identify relational tables by finding tables that have a subject column. We employ a simple heuristic for subject column detection: the subject column must be located in the first two columns of the table and contain unique entities. We further filter out tables containing less than three entities or more than twenty columns. With this process, we obtain 670,171 relational tables.

Data partitioning. We further select a high quality subset of the relational tables for evaluation: From tables that have (1) more than four linked entities in the subject column, (2) at least three entity columns including the subject column, and (3) more than half of the cells in entity columns are linked. We randomly select 10000 to form a held-out set, and partition it into validation/test sets via a rough 1:1 ratio for model evaluation. All relational tables not in the evaluation set are used for pre-training. In sum, we have 570171 / 5036 / 4964 tables respectively for pre-training/validation/test sets.

Most tables in our pre-training dataset have moderate size, with an average of 13 rows and 2 entity columns. We use the token vocabulary from BERT [13], and construct the entity vocabulary based on entities that appear more than once in the pre-training table corpus (926,135 entities).

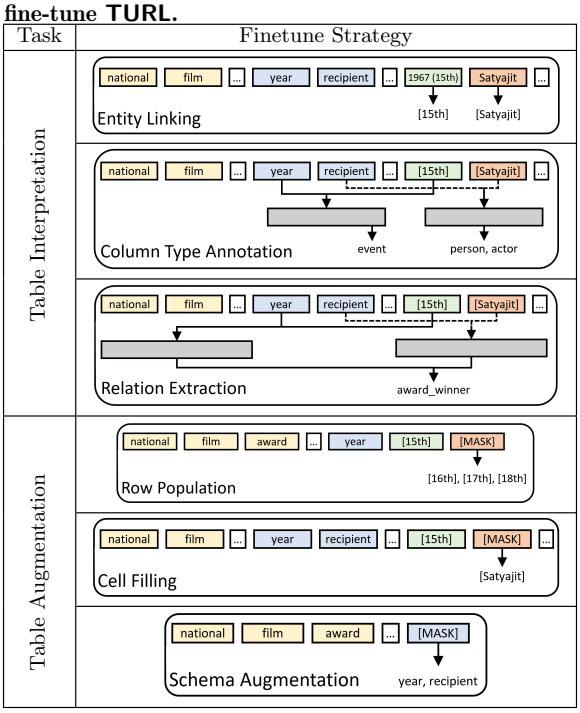
5. EXPERIMENTS

To systematically evaluate our pre-trained framework as well as facilitate research, we compile a table understanding benchmark consisting of 6 widely studied tasks covering table interpretation and table augmentation. Our pre-trained framework is general and can be applied to all the tasks with minimal task-specific fine-tuning.

5.1 Entity Linking

Definition. Given a table T and a knowledge base $\mathcal{KB}$, entity linking aims to link each potential mention in cells of T to its referent entity $e \in \mathcal{KB}$. Here we consider $\mathcal{KB}$ as structured databases that store descriptions and relations of entities in structured formats such as RDF triples.

Table 1: An overview of our benchmark tasks and strategies to fine-tune TURL.



Fine-tuning TURL. For entity linking, we first generate candidate entities using the Wikidata Lookup service. We then rank them by their matching scores with the target cell. We obtain a contextualized representation $\mathbf{h}^{\mathrm{e}}$ for each cell with its cell text (i.e., entity mention $\mathbf{e}^{\mathrm{m}}$ in Eqn. 2) and table metadata. To represent each candidate entity, we utilize the name, description and type information from a KB. Specifically, for a KB entity e, given its name N and description D (both are a sequence of words) and types T, we get its representation $\mathbf{e}^{\mathrm{kb}}$ as follows:

$$\mathbf{e}^{\mathrm{kb}} = [\texttt{MEAN}_{w \in N}\left(\mathbf{w}\right), \texttt{MEAN}_{w \in D}\left(\mathbf{w}\right), \texttt{MEAN}_{t \in T}\left(\mathbf{t}\right)]. \quad (7)$$

Here, $\mathbf{w}$ is the embedding for word w, and $\mathbf{t}$ is the embedding for entity type t. We then calculate a matching score between $\mathbf{e}^{\mathrm{kb}}$ and $\mathbf{h}^{\mathrm{e}}$ similarly as Eqn. 5. We do not use the entity embeddings pre-trained by our model here, as the goal is to link mentions to entities in a target KB, not necessarily those appear in our pre-training table corpus. The model is fine-tuned with a cross-entropy loss.

Datasets and baselines. We test with three datasets: (1) The Wikipedia gold standards (WikiGS) dataset [14], (2) our own test set from the held-out test tables mentioned in Section 4.4, (3) the T2D dataset [21] with Web tables from websites other than Wikipedia. We use names and descriptions returned by Wikidata Lookup, and entity types from DBpedia. We compare against the most recent methods for table entity linking T2K [26], Hybrid II [14], as well as the off-the-shelf Wikidata Lookup service.

Results. Results are shown in Table 2. Our model gets the best F1 score and substantially improves precision on WikiGS and our own test set. The disambiguation accuracy on WikiGS is 89.62% (predict the correct entity if it is in the candidate set). A more advanced candidate generation module can help achieve better results in the future. On the T2D dataset, all models perform much better than on the two Wikipedia datasets, mainly because of its smaller size and limited types of entities. The Wikidata Lookup baseline achieved high performance, and re-ranking using our model does not further improve. However, we adopt simple reweighting[2] to take into account the original result returned by Wikidata Lookup, which brings the F1 score to 0.82. This demonstrates the potential of using features such as entity popularity (used in Wikidata Lookup) and ensembling strong base models.

Table 2: Model evaluation on entity linking. We use the same TURL + fine-tuning model for all three datasets.

Method	WikiGS			Our Test Set			T2D		
	F1	P	R	F1	P	R	F1	P	R
T2K [26]	34	70	22	-	-	-	82	**90**	76
Hybrid II [14]	64	69	**60**	-	-	-	**83**	85	**81**
Wikidata Lookup	57	67	49	62	62	60	80	86	75
TURL + fine-tuning	**67**	**79**	58	**68**	**71**	**66**	78	83	73
+ reweighting	-	-	-	-	-	-	82	88	77

Table 3: Model evaluation on column type annotation.

Method	F1	P	R
Sherlock (only entity mention) [17]	78.47	88.40	70.55
TURL + fine-tuning (only entity mention)	88.86	90.54	87.23
TURL + fine-tuning	**94.75**	**94.95**	**94.56**
w/o table metadata	93.77	94.80	92.76
only table metadata	90.24	89.91	90.58

5.2 Column Type Annotation

Definition. Given a table T and a set of semantic types $\mathcal{L}$, column type annotation refers to the task of annotating a column in T with $l \in \mathcal{L}$ so that all entities in the column have type l. Note that a column can have multiple types.

Earlier work [26, 36] often coupled column type annotation with entity linking. More recently, [6, 7, 17] have studied column type annotation based on cell texts only. Here we adopt a similar setting, i.e., use the available information in a given table directly for column type annotation without performing entity linking first.

Fine-tuning TURL. We calculate the probability of predicting type l given column $\mathbf{h}_c$ as follows:

$$\mathbf{h}_c = [\texttt{MEAN}\left(\mathbf{h}_i^{\mathrm{t}}, \dots\right); \texttt{MEAN}\left(\mathbf{h}_j^{\mathrm{e}}, \dots\right)]; \quad (8)$$

$$P(l) = \mathrm{Sigmoid}\left(\mathbf{h}_c W_l + b_l\right). \quad (9)$$

Here $\mathbf{h}_i^{\mathrm{t}}$'s are representations of tokens in the column header, $\mathbf{h}_j^{\mathrm{e}}$'s are representations of entity cells in the column. We optimize the model with binary cross-entropy loss.

Datasets and baselines. We refer to Freebase [15] to obtain semantic types $\mathcal{L}$. For each column, we use the common types of its entities in Freebase as annotations. We further filter columns with less than three linked entities and keep only the most representative types. In the end, we get a total number of 255 types, 628,254 columns from 397,098 tables for training, 13,025 (13,391) columns from 4,764 (4,844) tables for testing (validation). We also test our model on two existing small-scale datasets, T2D-Te and Efthymiou [7]. Follow the setting in [7], we use 70% of T2D as training data (250 columns). We compare our results with Sherlock [17] and HNN + P2Vec [7].

[2]We simply reweight the score of the top-1 prediction by our model with a factor of 0.8 and compare it with the top-1 prediction returned by Wikidata Lookup. The higher one is chosen as final prediction.

Table 4: Accuracy on T2D-Te and Efthymiou for column type annotation.

Method	T2D-Te	Efthymiou
HNN + P2Vec (entity mention + KB) [7]	0.966	0.650
TURL + fine-tuning (only entity mention)	0.940	0.516
+ table metadata	0.962	0.746

Table 5: Model evaluation on relation extraction.

Method	F1	P	R
BERT-based	90.94	91.18	90.69
TURL + fine-tuning (only table metadata)	92.13	91.17	93.12
TURL + fine-tuning	**94.91**	**94.57**	**95.25**
w/o table metadata	93.85	93.78	93.91

Results. Results are shown in Table 3. Our model substantially outperforms the baselines. Further analysis for several types shows that although all methods work well for coarse-grained types like `person`, using table metadata greatly improves the results for fine-grained types like `actor` and `pro_athlete`. This indicates the importance of table context for predicting fine-grained column types. Results on T2D-Te and Efthymiou are summarized in Table 4. We can see that without using KB information, our model with entity mention and table metadata still outperforms or is on par with the baseline. However, when using only entity mention, our model does not perform as well as the baseline, especially when generalizing to Efthymiou. This is because: (1) Our model is pretrained with table metadata and entity embedding. Removing both creates a big mismatch between pretraining and fine-tuning. (2) With only 250 training instances, it is easy for deep models to overfit.

5.3 Relation Extraction

Definition. Given a table T and a set of relations $\mathcal{R}$ in KB. For a subject-object column pair in T, we aim to annotate it with $r \in \mathcal{R}$ so that r holds between all entity pairs in the columns.

Most existing work [26, 36] infers relations between columns based on entity linking results. However, such methods rely on entity linking performance and suffer from KB incompleteness. Here we aim to conduct relation extraction without explicitly linking table cells to entities. *This is important as it allows the extraction of new knowledge from Web tables for tasks like knowledge base population.*

Fine-tuning TURL. We use similar model architecture as column type annotation as follows.

$$P(r) = \text{Sigmoid}\left([\mathbf{h}_c; \mathbf{h}_{c'}]W_r + b_r\right). \tag{10}$$

Here $\mathbf{h}_c, \mathbf{h}_{c'}$ are aggregated column representations obtained same as Eqn. 8. We optimize with binary cross-entropy loss.

Datasets and baselines. We prepare datasets for relation extraction in a similar way as column type annotation. Specifically, we obtain relations $\mathcal{R}$ from Freebase. For each table in our corpus, we pair its subject column with each of its object columns, and annotate the column pair with relations shared by more than half of the entity pairs in the columns. Finally, we obtain a total number of 121 relations, 62,954 column pairs from 52,943 tables for training, and 2072 (2,175) column pairs from 1467 (1,560) tables for testing (validation). We compare our model with a text based relation extraction model [37]. Here we adapt the setting by treating the concatenated table metadata as a sentence, and the headers of the two columns as entity mentions. We also implement an entity linking based system using our entity linking model described in Section 5.1, and predict a relation if it holds between a minimum portion of linked entity pairs in KB.

Table 6: Relation extraction results of an entity linking based system, under different agreement ratio thresholds.

Min Ag. Ratio	F1	P	R
0	68.73	60.33	**79.85**
0.4	82.10	94.65	72.50
0.7	63.10	99.37	46.23

Table 7: Model evaluation on row population. Recall is the same for all methods because they share the same candidate generation module.

# seed	0		1	
Method	MAP	Recall	MAP	Recall
EntiTables [33]	17.90	63.30	42.31	78.13
Table2Vec [9]	-	63.30	20.86	78.13
TURL + fine-tuning	**40.92**	63.30	**48.31**	78.13

Results. From the main results in Table 5, we can see that our model outperforms the BERT-based baseline under all settings. Moreover, we notice that our model converges much faster in comparison to the BERT-based baseline during training, demonstrating that our model learns a better initialization through pre-training. Results for the entity linking based system are summarized in Table 6. We can see that it can achieve high precision, but suffers from low recall: The upper-bound of recall is only 79.85%, achieved at an agreement ratio of 0 (i.e., taking all relations that exist between the linked entity pairs as positive). As seen from Table 5 and 6, our model also substantially outperforms the system based on a strong entity linker.

5.4 Row Population

Definition. Given a partial table T, and an optional set of seed subject entities, row population aims to retrieve more entities to fill the subject column.

Fine-tuning TURL. We first generate candidate entities follow [33], which formulates a search query using either the table caption or seed entities and then retrieves tables via the BM25 retrieval algorithm. Subject entities in those retrieved tables will be candidates for row population. We then append the [MASK] token to the input, and use the hidden representation $\mathbf{h}^e$ of [MASK] to rank these candidates as shown in Table 1. We fine-tune our model with multi-label soft margin loss.

Datasets and baselines. We use tables in our pre-training set that have more than 3 subject entities for fine-tuning TURL and developing baseline models, and use tables in our held-out set with more than 5 subject entities for evaluation. In total, we obtain 432,660 tables for fine-tuning with 10 subject entities on average, and 4,132 (4,205) tables for testing (validation) with 16 (15) subject entities on average. We adopt models from [33] and [9] as baselines.

Results. We experiment both with and without seed entity. As shown in Table 7, our method outperforms all baselines. In particular, previous methods rely on entity similarity and are not applicable or have poor results when there is no seed entity available. Our method achieves a decent performance even without any seed entity, which demonstrates the effectiveness of TURL for generating contextualized representations based on both table metadata and content.

Table 8: Precision @ K on cell filling.

Method	P @ 1	P @ 3	P @ 5	P @ 10
Exact	51.36	70.10	76.80	84.93
H2H	51.90	70.95	77.33	85.44
H2V	52.23	70.82	77.35	85.58
TURL	**54.80**	**76.58**	**83.66**	**90.98**

Table 9: Mean Average Precision on schema augmentation.

Method	#seed column labels	
	0	1
kNN	80.16	**82.01**
TURL + fine-tuning	**81.94**	77.55

5.5 Cell Filling

Definition. Given a partial table T with the subject column filled and an object column header, cell filling aims to predict the object entity for each subject entity.

Fine-tuning TURL. Since cell filling is very similar to the MER pre-training task, we do not fine-tune the model, and directly use [MASK] to select from candidate entities same as MER (Eqn. 5). We use the same candidate generation module for TURL and baselines, which we describe in the next section.

Datasets and baselines. To evaluate different methods on this task, we use the held-out test tables in our pre-training phase and extract from them those subject-object column pairs that have at least three valid entity pairs. Finally we obtain 9,075 column pairs for evaluation. We adopt [34] as our base model. It has two main components, candidate value finding and value ranking. The same candidate value finding module is used for all methods: Given a subject entity e and object header h for the to-be-filled cells, we find all entities that appear in the same row with e in our pre-training table corpus, and only keep entities whose source header h' is related to h. Here we use the formula from [34] to measure the relevance of two headers $P(h'|h)$,

$$P(h'|h) = \frac{n(h', h)}{\sum_{h''} n(h'', h)}. \tag{11}$$

Here $n(h', h)$ is the number of table pairs in the table corpus that contain the same entity for a given subject entity in columns h' and h. We can then get the probability of the candidate entity e belongs to the cell $P(e|h)$ as follows:

$$P(e|h) = \text{MAX}\left(\text{sim}(h', h)\right). \tag{12}$$

Here h''s are the source headers associated with the candidate entity in the pre-training table corpus. $\text{sim}(h', h)$ is the similarity between h' and h. We develop three baseline methods for $\text{sim}(h', h)$: (1) **Exact**: predict the entity with exact matched header, (2) **H2H**: use the $P(h'|h)$ described above. (3) **H2V**: similar to [9], we train header embeddings with Word2Vec on the table corpus. We then measure the similarity between headers using cosine similarity.

Results. We use the same candidate value finding module for all methods, which obtains 61.45% recall with an average of 86 candidates. For value ranking, we only consider those test instances with the target object entity in the candidate set and evaluate them under Precision@K (or, P@K). From Table 8 we can see that: (1) Simple **Exact** match achieves decent performance, and using **H2H** or **H2V** only sightly improves the results. (2) Even though our model directly ranks the candidate entities without explicitly using their source table information, it outperforms other methods. This indicates that our model already encodes the factual knowledge in tables into entity embeddings through pre-training.

5.6 Schema Augmentation

Definition. Given a partial table T, which has a caption and zero or a few seed headers, and a header vocabulary $\mathcal{H}$, schema augmentation aims to recommend a ranked list of headers $h \in \mathcal{H}$ to add to T.

Fine-tuning TURL. We concatenate the table caption, seed headers and a [MASK] token as input to our model. The output for [MASK] is then used to predict the headers in a given header vocabulary $\mathcal{H}$. We fine-tune our model use binary cross-entropy loss.

Datasets and baselines. We collect $\mathcal{H}$ from the pre-training table corpus. We normalize the headers using simple rules, only keep those that appear in at least 10 different tables, and finally obtain 5652 unique headers, with 316,858 training tables and 4,646 (4,708) test (validation) tables. We adopt the method in [33] (kNN) which searches our pre-training table corpus for related tables, and use headers in those related tables for augmentation.

Results. From Table 9, we observe that both kNN baseline and our model achieve good performance. Our model works better when no seed header is available, but does not perform as well when there is one seed header. Further analysis shows that: One major reason why kNN works well is that there exist tables in the pre-training table corpus that are very similar to the query table and have almost the same table schema. On the other hand, our model oftentimes suggests plausible, semantically related headers, but misses the ground-truth headers.

6. CONCLUSION AND FUTURE WORK

We present TURL, a novel framework for learning deep contextualized representations on relational Web tables via self-supervised pre-training and task-specific fine-tuning. We propose a structure-aware Transformer encoder to model the row-column structure, and a new Masked Entity Recovery objective to capture the relational knowledge in tables. To facilitate research in this direction, we compile a new benchmark that consists of 6 different tasks for table understanding and conduct comprehensive experiments.

This paper, for the first time to our knowledge, studies the application of the pre-training/fine-tuning paradigm on relational table understanding. Although we mainly focus on relational Web tables, our techniques could be extended to other (semi-)structured data formats, such as database tables, SpreadSheets, JSON etc. In this work, we examine the utility of our approach for table interpretation and table augmentation. Some recent work also researches the potential of the pre-training/fine-tuning paradigm on other tasks such as table searching [29], table to text generation [30] and building natural language interfaces [10, 32]. More broadly, we believe that developing more advanced machine learning techniques to understand (semi-)structured data could benefit all stages of data management, from data preparation to data query and analysis. We hope our work could inspire more future work along that direction.

Acknowledgments: Authors at the Ohio State University were sponsored in part by Google Faculty Award, the Army Research Office under cooperative agreements W911NF-17-

1-0412, NSF Grant IIS1815674, NSF OAC-2112606, NSF CAREER #1942980, Fujitsu gift grant, and Ohio Supercomputer Center [5]. The views and conclusions contained herein are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Office or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notice herein.

7. REFERENCES

[1] A. Ahmadov, M. Thiele, J. Eberius, W. Lehner, and R. Wrembel. Towards a hybrid imputation approach using web tables. In *BDC*, 2015.

[2] C. S. Bhagavatula, T. Noraset, and D. Downey. Tabel: Entity linking in web tables. In *ISWC*, 2015.

[3] M. J. Cafarella, A. Halevy, D. Z. Wang, E. Wu, and Y. Zhang. Webtables: exploring the power of tables on the web. *VLDB*, 2008.

[4] M. J. Cafarella, A. Y. Halevy, Y. Zhang, D. Z. Wang, and E. Wu. Uncovering the relational web. In *WebDB*, 2008.

[5] O. S. Center. Ohio supercomputer center, 1987.

[6] J. Chen, E. Jiménez-Ruiz, I. Horrocks, and C. A. Sutton. Colnet: Embedding the semantics of web tables for column type prediction. In *AAAI*, 2018.

[7] J. Chen, E. Jiménez-Ruiz, I. Horrocks, and C. A. Sutton. Learning semantic annotations for tabular data. In *IJCAI*, 2019.

[8] A. Das Sarma, L. Fang, N. Gupta, A. Halevy, H. Lee, F. Wu, R. Xin, and C. Yu. Finding related tables. In *SIGMOD*, 2012.

[9] L. M. Deng, S. Zhang, and K. Balog. Table2vec: Neural word and entity embeddings for table population and retrieval. In *SIGIR*, 2019.

[10] X. Deng, A. Hassan, C. Meek, O. Polozov, H. Sun, and M. Richardson. Structure-grounded pretraining for text-to-sql. In *NAACL*, 2021.

[11] X. Deng, H. Sun, A. Lees, Y. Wu, and C. Yu. Turl: Table understanding through representation learning. *VLDB*, 2020.

[12] X. Deng, H. Sun, A. Lees, Y. Wu, and C. Yu. Turl: Table understanding through representation learning. `https://arxiv.org/abs/2006.14806`, 2020.

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

[14] V. Efthymiou, O. Hassanzadeh, M. Rodriguez-Muro, and V. Christophides. Matching web tables with knowledge base entities: From entity lookups to entity embeddings. In *ISWC*, 2017.

[15] Google. Freebase data dumps. `https://developers.google.com/freebase/data`.

[16] J. Herzig, P. Nowak, T. Müller, F. Piccinno, and J. M. Eisenschlos. Tapas: Weakly supervised table parsing via pre-training. In *ACL*, 2020.

[17] M. Hulsebos, K. Z. Hu, M. Bakker, E. Zgraggen, A. Satyanarayan, T. Kraska, cCaugatay Demiralp, and C. A. Hidalgo. Sherlock: A deep learning approach to semantic data type detection. *KDD*, 2019.

[18] X. Jiao, Y. Yin, L. Shang, X. Jiang, X. Chen, L. Li, F. Wang, and Q. Liu. Tinybert: Distilling bert for natural language understanding. *ArXiv*, 2019.

[19] E. Jiménez-Ruiz, O. Hassanzadeh, V. Efthymiou, J. Chen, and K. Srinivas. Semtab 2019: Resources to benchmark tabular data to knowledge graph matching systems. In *ESWC*, 2020.

[20] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *ArXiv*, 2015.

[21] O. Lehmberg, D. Ritze, R. Meusel, and C. Bizer. A large public corpus of web tables containing time and context metadata. In *WWW*, 2016.

[22] Y. Li, J. Li, Y. Suhara, A. Doan, and W.-C. Tan. Deep entity matching with pre-trained language models. *ArXiv*, 2020.

[23] G. Limaye, S. Sarawagi, and S. Chakrabarti. Annotating and searching web tables using entities, types and relationships. In *VLDB*, 2010.

[24] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. In *NeurIPS*, 2013.

[25] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever. Language models are unsupervised multitask learners. 2019.

[26] D. Ritze, O. Lehmberg, and C. Bizer. Matching html tables to dbpedia. In *WIMS*, 2015.

[27] Q. shan Wang, Z. Mao, B. Wang, and L. Guo. Knowledge graph embedding: A survey of approaches and applications. *TKDE*, 2017.

[28] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. In *NeurIPS*, 2017.

[29] F. Wang, K. Sun, M. Chen, J. Pujara, and P. Szekely. Retrieving complex tables with multi-granular graph representation learning. *SIGIR*, 2021.

[30] T. Xie, C. H. Wu, P. Shi, R. Zhong, T. Scholak, M. Yasunaga, C.-S. Wu, M. Zhong, P. Yin, S. I. Wang, V. Zhong, B. Wang, C. Li, C. Boyle, A. Ni, Z. Yao, D. Radev, C. Xiong, L. Kong, R. Zhang, N. A. Smith, L. Zettlemoyer, and T. Yu. Unifiedskg: Unifying and multi-tasking structured knowledge grounding with text-to-text language models. *ArXiv*, 2022.

[31] P. Yin, G. Neubig, W. tau Yih, and S. Riedel. Pretraining for joint understanding of textual and tabular data. In *ACL*, 2020.

[32] T. Yu, C.-S. Wu, X. V. Lin, Y. C. Tan, X. Yang, D. Radev, C. Xiong, et al. Grappa: Grammar-augmented pre-training for table semantic parsing. In *ICLR*, 2020.

[33] S. Zhang and K. Balog. Entitables: Smart assistance for entity-focused tables. In *SIGIR*, 2017.

[34] S. Zhang and K. Balog. Auto-completion for data cells in relational tables. In *CIKM*, 2019.

[35] S. Zhang and K. Balog. Web table extraction, retrieval, and augmentation: A survey. *TIST*, 2020.

[36] Z. Zhang. Effective and efficient semantic table interpretation using tableminer+. *Semantic Web*, 2017.

[37] Z. Zhang, X. Han, Z. Liu, X. Jiang, M. Sun, and Q. Liu. Ernie: Enhanced language representation with informative entities. In *ACL*, 2019.

Technical Perspective – No PANE, No Gain: Scaling Attributed Network Embedding in a Single Server

Aidan Hogan
DCC, Universidad de Chile & IMFD
ahogan@dcc.uchile.cl

The machine learning community has traditionally been proactive in developing techniques for diverse types of data, such as text, audio, images, videos, time series, and, of course, matrices, tensors, etc. "*But what about graphs*?" some of us graph enthusiasts may have asked ourselves, dejectedly, before transforming our beautiful graph into a brutalistic table of numbers that bore little resemblance to its parent, nor the phenomena it represented, but could at least be shovelled into the machine learning frameworks of the time.

Thankfully those days are coming to an end.

The area of *graph representation learning* [1] has enjoyed growing momentum, spurred on not only by graph enthusiasts, but also by applications relating to transport networks, social networks, biological networks, and more recently, knowledge graphs. Within the area, we find a fork in the road. To the left, we find researchers developing novel learning frameworks over the topology of the graph itself, à la *graph neural networks*. Veering right, we find works on various forms of *graph* or *network embeddings* that transform elements of the graph into vectors, matrices, etc., that can be fed into the more traditional machine learning frameworks used for downstream tasks, but now in a principled and elegant way that is more respectful to the original graph and what it represents.

We can find two common limitations, however, when network embedding techniques are applied in real-world scenarios. First, they may struggle to cope with large graphs (let's say in the order of billions of edges). Second, in order to learn representations of more complex graph data, we often need to learn representations for features like edge direction, edge labels, node attributes, etc.

The paper "*No PANE, No Gain: Scaling Attributed Network Embedding in a Single Server*" by Yang et al. [2] addresses these two key issues of *scalability* and handling *complex features* in the context of network embeddings. The model they adopt is that of an *attributed network*, which features directed edges, where nodes can additionally be associated with weighted attributes. This approach thus narrows the gap between network embedding techniques that typically address undirected graphs and graph models popular in practice with features like directed edges and node attributes. An attributed network could be used to model, for example, a citation graph, where nodes represent edges, directed edges indicate citations, and attributes indicate (weighted) topics for each paper.

The stated goal of the paper is then to learn vector representations of nodes within an attributed network that capture "node–attribute affinities", i.e., attributes reachable from a node through zero or more hops in the graph. Both forward and backward affinities are considered. This can capture, for example, what affinities a paper (a node) has with which topics (attributes) based not only on its own topics, but also those of the papers it cites, and the paper they cite, etc., (forward affinities); as well as the topics of the papers that cite it, and the papers they are cited by, etc. (backwards affinities). These affinities are captured through forward and backward random walks (with restart) that explore neighbours in both directions with a given stop probability at each step, recording a random attribute of the node where the walk stops. Two vectors – for forwards and backwards affinity – are then learnt for each node, and one vector is learnt for each attribute, such that the dot product of a given forward-affinity node vector and a given attribute vector approximate the corresponding affinity seen through random walks; the same is applied for backwards affinity.

The pressing challenge now is to train vectors at large scale, where it is computationally challenging, for example, to jointly optimise both forwards and backwards affinities together. The paper thus proposes two novelties to address this issue. The first is to approximate both forward and backward affinities by replacing the random walks with a fixed number of iterations of a power method-style algorithm to approximate the target affinities in both directions. The second is to propose an efficient method for computing the necessary embeddings that first applies a matrix factorization of the forward and backward affinity matrices produced by the previous step in order to approximate the corresponding embeddings separately, and using these embeddings to initialise a cyclic coordinate descent method, which then jointly optimises the embeddings according to both forward and backward affinities. Finally, the authors show how the proposed algorithm, which they call PANE, can be effectively parallelised over multiple cores.

Experiments compare PANE with a wide range of baselines on a wide range of graphs, for a variety of downstream tasks, including attribute inference (predicting the attributes of a node), link prediction (predicting which node a give node links to), and node classification (predicting a label for a node). PANE is shown to largely outperform the competitors in terms not only of quality metrics (such as average precision), but also in terms of scale and performance, with only one other competitor able to cope with the largest graph considered: the Microsoft Academic Graph, with 59 million nodes, 2 thousand attributes, and close to a billion edges.

The paper thus advances the state of the art in a number of ways that should please the average graph enthusiast, showing not only how node attributes can be taken into account when learning embeddings, but also how such an approach can be made to scale by using approximations, clever initialisation techniques, and parallelism. A pressing question for the future then: how could these techniques be adapted to support attributes *and* edge labels?

REFERENCES

[1] William L. Hamilton. 2020. *Graph Representation Learning*. Morgan & Claypool Publishers. https://doi.org/10.2200/S01045ED1V01Y202009AIM046
[2] Renchi Yang, Jieming Shi, Xiaokui Xiao, Yin Yang, Juncheng Liu, and Sourav S. Bhowmick. 2020. Scaling Attributed Network Embedding to Massive Graphs. *Proc. VLDB Endow.* 14, 1 (2020), 37–49. https://doi.org/10.14778/3421424.3421430

No PANE, No Gain: Scaling Attributed Network Embedding in a Single Server

Renchi Yang*
National University of Singapore
renchi@nus.edu.sg

Jieming Shi
Hong Kong Polytechnic University
jieming.shi@polyu.edu.hk

Xiaokui Xiao
National University of Singapore
xkxiao@nus.edu.sg

Yin Yang
Hamad bin Khalifa University
yyang@hbku.edu.qa

Sourav S. Bhowmick
Nanyang Technological University
assourav@ntu.edu.sg

Juncheng Liu
National University of Singapore
juncheng@comp.nus.edu.sg

ABSTRACT

Given a graph G where each node is associated with a set of attributes, *attributed network embedding* (*ANE*) maps each node $v \in G$ to a compact vector X_v, which can be used in downstream machine learning tasks in a variety of applications. Existing ANE solutions do not scale to massive graphs due to prohibitive computation costs or generation of low-quality embeddings. This paper proposes `PANE`, an effective and scalable approach to ANE computation for massive graphs in a *single* server that achieves state-of-the-art result quality on multiple benchmark datasets for two common prediction tasks: link prediction and node classification. Under the hood, `PANE` takes inspiration from well-established data management techniques to scale up ANE in a single server. Specifically, it exploits a carefully formulated problem based on a novel random walk model, a highly efficient solver, and non-trivial parallelization by utilizing modern multi-core CPUs. Extensive experiments demonstrate that `PANE` consistently outperforms all existing methods in terms of result quality, while being orders of magnitude faster.

1. INTRODUCTION

Graphs (a.k.a networks) are ubiquitous nowadays in many application domains such as biology, social sciences, chemistry, and finance. A recent survey [12] revealed that scalability and faster graph analytics or machine learning (ML) algorithms are considered as some of the top challenges for graph processing. Although considerable efforts have been invested toward these goals in academia and industry, these issues remain tenaciously challenging to address due to high computational complexity of iterative or combinatorial graph algorithms, low parallelizability due to tight coupling between nodes and edges, and difficulty to leverage traditional graph representation (*e.g.*, adjancency matrix) for ML problems [5]. In particular, many ML techniques typically assume independent real-valued input vectors and outputs in order to learn a latent function that maps each input to an output. Unfortunately, nodes in any network data are coupled through their edges, making it challenging for using traditional network representations in ML techniques. Although in principle we can represent the nodes as their corresponding row vectors in the adjacency matrix of the network, the dimensionalilty of such simplistic representation can be prohibitively large, rendering them impractical.

Network Embedding. Network embedding [5] aims to address the aforementioned challenges of traditional network representations by learning low-dimensional, fixed-length vector representations of network nodes such that the similarity in the embedding space reflects the similarity in the network. Specifically, in the original network, relationships between nodes are captured by edges or other higher-order topological measures. In the embedding space, these relationships are captured by distances between nodes in the vector space and the topological properties of the nodes are encoded by their corresponding embedding vectors. Since each node is represented by a vector encapsulating information of interest, many iterative or combinatorial graph problems can be reframed as computing mapping functions and operations on the embedding vectors, paving the way for more efficient or scalable solutions. Furthermore, the learned embedding space enables various network inference tasks such as link prediction, node label inference, and finding "important" nodes. For example, we can input two real-valued vectors representing a pair of nodes to a machine learning algorithm to predict the existence of a link between them (*i.e.*, a binary classification problem where the output label of 0 or 1 represents absence or presence of a link, respectively). Observe that the learned embeddings for all these tasks are realizable without demanding expensive feature engineering by domain experts. All these opportunities have led to the proposal of a cornucopia of techniques in the literature for network embedding and their usage in a wide variety of applications [5].

Attributed Network Embedding. The majority of existing network embedding techniques, however, exploit only the topological connections when learning node representations. In practice, however, real-world networks often are *attributed*

*Work done when the first author was a doctoral student at the Nanyang Technological University, Singapore.

DOI: https://doi.org/10.14778/3421424.3421430 .

networks where nodes are associated with attribute-value pairs that capture important information about them. Techniques that are oblivious to such rich information associated with nodes often tend to learn poorer quality node representations, adversely impacting downstream ML tasks [14]. For example, consider a pair of users, u_1 and u_2, in a social network who are in topologically close proximity. Assume that u_1 is interested in the game of cricket whereas u_2's interest lies in canoeing. Ignoring such attribute information of u_1 and u_2 by simply considering only their neighborhoods' structural features may lead to an inferior-quality vector space representation of them. As an aftermath, a link may be predicted between u_1 and u_2 due to their topological similarity although they are individuals with highly dissimilar taste. Furthermore, the information associated with node attributes are even more useful in sparse scale-free networks where such information can complement scant topological information for learning superior embedding vectors.

Attributed network embedding (ANE) [14] aims to map *both* topological and attribute information surrounding a node to an embedding vector to facilitate superior network inference tasks. At first glance, it may seem that we can treat topology and attributes as separate features to address the ANE problem. Unfortunately, doing so loses the important information of *node-attribute affinity* [9], *i.e.*, attributes that can be reached by a node through one or more hops along the edges in the network. Hence, the key challenge to address the ANE problem is to devise efficient and scalable ways to *integrate* these information for network embedding.

Research Challenges and Gap in ANE. Effective ANE computation is a highly challenging task, especially for massive graphs. In particular, each node v in a network G could be associated with a large number of attributes, adding up to the number of dimensions. Furthermore, each attribute of v could influence not only v's own embedding, but also those of v's neighbors, neighbors' neighbors, and far-reaching connections via multiple hops along the edges in G.

Existing ANE solutions can be broadly classified into two categories, *factorization-based* and *auto-encoder-based* approaches. Unfortunately, these solutions are prohibitively expensive and largely fail on massive networks. Factorization-based methods [14–16] are based on the idea of reducing an $n \times n$ matrix, where n is the number of nodes in G, into its smaller constituent parts so that embeddings can be discovered from the latter. In order to realize this, the $n \times n$ matrix often needs to be explicitly constructed and factorized. For a graph with 50 million nodes, storing such a matrix of double-precision values would require over 20 petabytes of memory, which is clearly infeasible. On the other hand, auto-encoder-based strategies [8, 9, 11] employ deep neural networks to extract higher-order features from nodes' connections and attributes. For a large dataset, training such a neural network incurs vast computational costs. In addition, the training process is usually done on GPUs with limited graphics memory. Consequently, for massive graphs, currently the only option is to compute ANE leveraging a large cluster, which is rather expensive, and has a significant environmental impact.

In addition, many existing ANE solutions are designed for undirected graphs. In reality, directed networks are common; as we shall see later, these methods yield suboptimal result quality on directed networks.

Gain with PANE. In this paper, we provide an affirmative answer to the following question of significance: *Can we efficiently compute effective ANE embeddings on a massive, attributed, directed graph on a single server?* To this end, we present `PANE`, a novel solution that significantly advances the state of the art in ANE computation. The key idea behind our solution is to devise techniques inspired by established ideas used in data management to speed up and scale ANE operations. Specifically, `PANE` formulates ANE as an optimization problem with the objective of approximating *normalized multi-hop node-attribute affinity* using node-attribute co-projections [9], guided by a *shifted pairwise mutual information* (SPMI) metric. The affinity between a given node-attribute pair is defined via a novel random walk model with a flexible neighborhood sampling strategy specifically adapted to attributed networks. Further, we incorporate edge direction information by defining separate *forward* and *backward* affinity, embeddings, and SPMI metrics. Solving this optimization problem is still immensely expensive with off-the-shelf algorithms, as it involves the joint factorization of two $O(n \cdot d)$-sized matrices, where n and d are the numbers of nodes and attributes in the input data, respectively. Thus, `PANE` includes a novel solver with a key module that seeds the optimizer through a highly effective greedy algorithm, which drastically reduces the number of iterations till convergence. Finally, we devise database-inspired non-trivial parallelization of the `PANE` algorithm by utilizing modern multi-core CPUs judiciously without significantly compromising result quality.

Extensive experiments demonstrate that `PANE` consistently obtains high-utility embeddings with superior prediction accuracy for link prediction and node classification, at a fraction of the cost compared to existing methods. In particular, on the largest *Microsoft Academic Knowledge Graph* (*MAG*), `PANE` is the only viable solution on a single server, whose resulting embeddings lead to 0.965 AP for link prediction and 0.57 micro-F1[1] for node classification. Notably, it obtains these results using 10 CPU cores, 1TB memory, and within 12 hours running time.

Summary of Contributions. In summary, this paper makes the following contributions: (a) We formulate the ANE task as an optimization problem with the objective of approximating multi-hop node-attribute affinity. We consider edge direction in our objective by defining forward and backward affinity matrices using the SPMI metric. (b) We propose several techniques to efficiently solve the optimization problem, including efficient approximation of the affinity matrices, fast joint factorization of the affinity matrices, and a key module to greedily seed the optimizer, which drastically reduces the number of iterations till convergence. (c) We develop non-trivial parallelization techniques of `PANE` to further boost efficiency. (d) We experimentally demonstrate the superior performance of `PANE`, in terms of efficiency and effectiveness, against 10 competitors on 4 real datasets.

2. ATTRIBUTED NETWORK EMBEDDING

In this section, we formally introduce the notion of attributed network embedding (ANE) and discuss existing efforts to address this problem.

[1]The micro-F1 score, ranging from 0 to 1, is the harmonic mean of the precision and recall, which are computed through micro averaging [18].

Let $G = (V, E_V, R, E_R)$ be an *attributed network*, consisting of (i) a node set V with cardinality n, (ii) a set of m edges E_V, each connecting two nodes in V, (iii) a set of d attributes R, and (iv) a set of node-attribute associations E_R, where each element is a tuple $(v_i, r_j, w_{i,j})$ signifying that node $v_i \in V$ is directly associated with attribute $r_j \in R$ with weight $w_{i,j}$ (*i.e.*, attribute value). For example, given a user v_i in a social network and an attribute r_j representing age, weight $w_{i,j}$ denotes the value of v_i's age. Note that for a categorical attribute, we first transform it into a set of binary ones through one-hot encoding. Without loss of generality, we assume that G is a directed graph; if G is undirected, then we treat each edge (v_i, v_j) as a pair of directed edges with opposing directions: (v_i, v_j) and (v_j, v_i).

The *neighbohood* of a node v is typically generated using a graph traversal strategy such as breadth-first search, depth-first search, or a random walk. Intuitively, it represents a set of nodes that are in "close" proximity of v. Specifically, the first-order proximity indicates existence of links between a pair of nodes whereas higher-order proximity reflects the neighborhood. Note that neighborhood of different nodes can be overlapping and may be of different sizes.

Given a space budget $k \ll n$ and $k > 0$ (*i.e.*, dimensionality), a *node embedding* function $f : V \longrightarrow \mathbb{R}^k$ maps each node $v \in V$ to a length-k real-valued vector in $\mathbb{R}^k$. The broad goal of attributed network embedding (ANE) is to compute such an embedding X_v for each node v in the input graph, such that X_v captures the graph structure and attribute information of the neighborhood of v. Following previous work [9], we also allocate a space budget $\frac{k}{2}$ (detailed in Section 3.2) for each attribute $r \in R$, and aim to compute an *attribute embedding* vector for r of length $\frac{k}{2}$.

Related Work. Existing *factorization-based* methods [14–16] mainly involve two stages: (i) build an $n \times n$ *proximity matrix* that models the proximity between nodes based on graph topology or attribute information; (ii) factorize it via techniques such as stochastic gradient descent (SGD), alternating least square (ALS), and coordinate descent. As remarked earlier, all these methods incur immense overheads in building and factorizing the proximity matrix and are designed for undirected graphs only.

An auto-encoder is a neural network model consisting of an encoder that compresses the input data to obtain embeddings and a decoder that reconstructs the input data from the embeddings, with the goal of minimizing the reconstruction loss. Existing auto-encoder-based methods for ANE [8, 9, 11] either use proximity matrices as inputs or design various neural network structures for the auto-encoder. Typically, these methods suffer from severe efficiency issues due to the expensive training process of auto-encoders; further, none of them considers edge directions.

Lastly, there exist several techniques (*e.g.,* [13, 19]) that generate embeddings without matrix factorization or auto-encoder. They employ other expensive deep learning techniques, rendering them infeasible to handle massive graphs.

3. THE PANE ALGORITHM

This section presents the proposed PANE algorithm. We begin by introducing notations and terminology.Then, we describe the design principles and associated challenges in realizing PANE. Finally, we describe the sequential and parallel versions of the algorithm.

3.1 Terminology

We denote matrices in bold uppercase, *e.g.*, $\mathbf{M}$. We use $\mathbf{M}[v_i]$ to denote the v_i-th row vector of $\mathbf{M}$, and $\mathbf{M}[:, r_j]$ to denote the r_j-th column vector of $\mathbf{M}$. In addition, we use $\mathbf{M}[v_i, r_j]$ to denote the element at the v_i-th row and r_j-th column of $\mathbf{M}$. Given an index set S, we let $\mathbf{M}[S]$ (resp. $\mathbf{M}[:, S]$) be the matrix block of $\mathbf{M}$ that contains the row (resp. column) vectors of the indices in S.

We define an *attribute matrix* $\mathbf{R} \in \mathbb{R}^{n \times d}$, such that $\mathbf{R}[v_i, r_j] = w_{i,j}$ is the weight associated with the entry $(v_i, r_j, w_{ij}) \in E_R$. We refer to $\mathbf{R}[v_i]$ as node v_i's *attribute vector*. Based on $\mathbf{R}$, we derive a row-normalized (resp. column-normalized) attribute matrices $\mathbf{R}_r$ (resp. $\mathbf{R}_c$) as follows:

$$\mathbf{R}_r[v_i, r_j] = \frac{\mathbf{R}[v_i, r_j]}{\sum_{r_l \in R} \mathbf{R}[v_i, r_l]}, \ \mathbf{R}_c[v_i, r_j] = \frac{\mathbf{R}[v_i, r_j]}{\sum_{v_l \in V} \mathbf{R}[v_l, r_j]}. \quad (1)$$

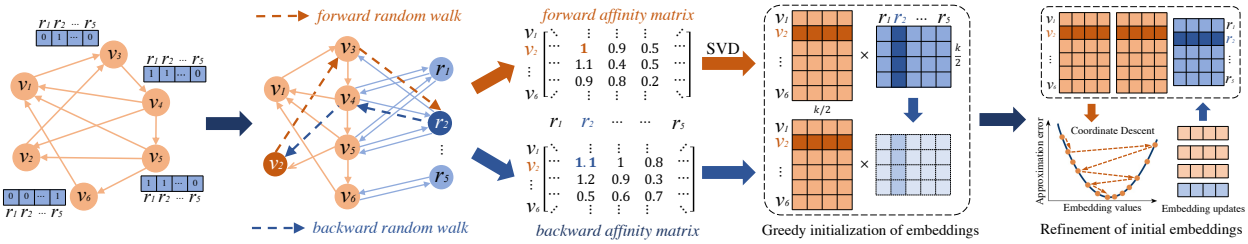
3.2 Design Principles and Challenges

Figure 1 depicts the PANE framework. Intuitively, we represent the input network in a way that is conducive for subsequent computation of *affinity* between node and attribute pairs. This information is subsequently exploited to generate the embeddings for each node. We elaborate on the design of these components and associated challenges.

Extended Graph. The broad goal here is to transform the attribute information associated with nodes in G to *special* nodes and edges to create a unified framework capturing topological and attribution information. To this end, we utilize the notion of *extended graph*, denoted by $\mathcal{G}$, which we explain with an example. Figure 1(ii) shows an example extended graph $\mathcal{G}$ constructed based on an input attributed network G (Figure 1(i)) consisting of 6 nodes v_1-v_6 and 5 attributes r_1-r_5. Observe that the nodes and edges in blue show the attribute associations E_R in G. Specifically, for each attribute $r_j \in R$, we create an additional node in $\mathcal{G}$; then, for each entry in E_R, *e.g.*, $(v_3, r_2, w_{3,2})$, we include in $\mathcal{G}$ a pair of edges with opposing directions connecting the node (*e.g.*, v_3) with the corresponding attribute node (*e.g.*, r_2), with an edge weight (*e.g.*, $w_{3,2}$). Note that in this example, nodes v_1 and v_2 are not associated with any attribute.

Forward Affinity and Backward Affinity. As remarked earlier, the resulting embedding of a node $v \in V$ should capture its *affinity* with attributes in R, where the affinity definition should take into account both the attributes directly associated with v in E_R, and the attributes of the nodes that v can reach via edges in E_V. To effectively model node-attribute affinity via multiple hops in $\mathcal{G}$, we employ an adaptation of the *random walks with restarts* (*RWR*) [7], a technique that has find successful usage in data management research for finding relevance score between two nodes [1, 7]. In the sequel, we refer to an RWR simply as a *random walk*. Specifically, since $\mathcal{G}$ is directed, we distinguish two types of node-attribute affinity: *forward affinity*, denoted as $\mathbf{F}$, and *backward affinity*, denoted as $\mathbf{B}$.

Given an attributed graph G, a node v_i, and random walk stopping probability α ($0 < \alpha < 1$), a *forward random walk* on $\mathcal{G}$ starts from node v_i. At each step, assume that the walk is currently at node v_l. Then, the walk can either (i) with probability α, terminate at v_l , or (ii) with probability $1-\alpha$, follow an edge in E_V to a random out-neighbor of v_l. After a random walk terminates at a node v_l, we randomly follow an edge in E_R to an attribute r_j, with probability $\mathbf{R}_r[v_l, r_j]$,



(i) Input graph G (ii) Extended graph $\mathcal{G}$ (iii) Construction of affinity matrices (iv) Joint factorization of affinity matrices

Figure 1: Overview of PANE

i.e., a normalized edge weight defined in Equation (1)[2]. The forward random walk yields a *node-to-attribute pair* (v_i, r_j), and we add this pair to a collection $\mathcal{S}_f$.

Suppose that we sample n_r node-to-attribute pairs for each node v_i, the size of $\mathcal{S}_f$ is then $n_r \cdot n$, where n is the number of nodes in G. Denote $p_f(v_i, r_j)$ as the probability that a forward random walk starting from v_i yields a node-to-attribute pair (v_i, r_j). Then, the *forward affinity* $\mathbf{F}[v_i, r_j]$ between note v_i and attribute r_j is defined as follows.

$$\mathbf{F}[v_i, r_j] = \log\left(\frac{n \cdot p_f(v_i, r_j)}{\sum_{v_h \in V} p_f(v_h, r_j)} + 1\right) \quad (2)$$

To explain the intuition behind the above definition, note that in collection $\mathcal{S}_f$, the probabilities of observing node v_i, attribute r_j, and pair (v_i, r_j) are $\mathbb{P}(v_i) = \frac{1}{n}$, $\mathbb{P}(r_j) = \frac{\sum_{v_h \in V} p_f(v_h, r_j)}{n}$, and $\mathbb{P}(v_i, r_j) = \frac{p_f(v_i, r_j)}{n}$, respectively. Thus, the above definition of forward affinity is a variant of the *pointwise mutual information*, (PMI)[3] [4] between node v_i and attribute r_j. Since PMI can be negative, we use the variant *shifted* PMI (SPMI), which is guaranteed to be positive. Hence, $\mathbf{F}[v_i, r_j]$ in Equation (2) is essentially SPMI(v_i, r_j).

We define backward affinity in a similar fashion. Given an attributed network G, an attribute r_j and stopping probability α, a *backward random walk* starting from r_j first randomly samples a node v_l according to probability $\mathbf{R}_c[v_l, r_j]$, defined in Equation (1). Then, the walk starts from node v_l and follows the aforementioned strategy. Suppose that the walk terminates at node v_i; then, it returns an *attribute-to-node pair* (r_j, v_i), which is added to a collection $\mathcal{S}_b$. After sampling n_r attribute-to-node pairs for each attribute, the size of $\mathcal{S}_b$ becomes $n_r \cdot d$. Let $p_b(v_i, r_j)$ be the probability that a backward random walk starting from attribute r_j stops at node v_i. In collection $\mathcal{S}_b$, the probabilities of observing attribute r_j, node v_i and pair (r_j, v_i) are $\mathbb{P}(r_j) = \frac{1}{d}$, $\mathbb{P}(v_i) = \frac{\sum_{r_h \in R} p_b(v_i, r_h)}{d}$ and $\mathbb{P}(v_i, r_j) = \frac{p_b(v_i, r_j)}{d}$, respectively. Then the *backward affinity* $\mathbf{B}[v_i, r_j]$ is as follows.

$$\mathbf{B}[v_i, r_j] = \log\left(\frac{d \cdot p_b(v_i, r_j)}{\sum_{r_h \in R} p_b(v_i, r_h)} + 1\right). \quad (3)$$

Objective Function. The above notions of forward and backward node-attribute affinity capture the necessary information from which we can learn the embeddings of each node. Specifically, given a space budget k, our goal is to learn (i) two embedding matrices $\mathbf{X}_f$, $\mathbf{X}_b \in \mathbb{R}^{n \times \frac{k}{2}}$ for all nodes in V, whose row vectors $\mathbf{X}_f[v_i] \in \mathbb{R}^{\frac{k}{2}}$ and $\mathbf{X}_b[v_i] \in \mathbb{R}^{\frac{k}{2}}$ denote the *forward embedding vector* and *backward embedding vector* for node v_i, respectively, and (ii) an embedding matrix $\mathbf{Y} \in \mathbb{R}^{d \times \frac{k}{2}}$ for all attributes in R, where each row vector $\mathbf{Y}[r_j] \in \mathbb{R}^{\frac{k}{2}}$ is the *attribute embedding vector* for attribute r_j. Mathematically, we can express this objective as to learn $\mathbf{X}_f, \mathbf{X}_b$ and $\mathbf{Y}$ such that the following objective is minimized:

$$\mathcal{O} = \min_{\mathbf{X}_f, \mathbf{Y}, \mathbf{X}_b} \sum_{v_i \in V} \sum_{r_j \in R} \left(\mathbf{F}[v_i, r_j] - \mathbf{X}_f[v_i] \cdot \mathbf{Y}[r_j]^\top\right)^2 + \left(\mathbf{B}[v_i, r_j] - \mathbf{X}_b[v_i] \cdot \mathbf{Y}[r_j]^\top\right)^2. \quad (4)$$

Intuitively, we approximate the two affinities between node v_i and attribute r_j using the dot product of their respective embedding vectors. The objective is then to minimize the total squared error of such approximations, over all nodes and all attributes in the input data.

Challenges. A keen reader may observe that it is prohibitively expensive to train embeddings of nodes and attributes that preserve our objective function in Equation (4), especially on massive attributed networks. First, node-attribute affinity values are defined by random walks, which are rather expensive to conduct in a huge number from every node and attribute of massive graphs. Second, our objective function preserves both forward and backward affinity (*i.e.*, it takes into account edge directions), which makes the training process hard to converge. Further, jointly preserving both forward and backward affinity involves intensive computations, severely dragging down the performance. We tackle these challenges in the next subsections.

3.3 Seq-PANE: A Sequential Algorithm

Intuitively, PANE consists of three phases: (i) iteratively computing approximated versions $\mathbf{F}'$ and $\mathbf{B}'$ of the forward and backward affinity matrices with rigorous approximation error guarantees, without actually sampling random walks, (ii) initializing the embedding vectors with a greedy algorithm for fast convergence, and then (iii) jointly factorizing $\mathbf{F}'$ and $\mathbf{B}'$ using *cyclic coordinate descent* to efficiently obtain the embedding vectors $\mathbf{X}_f, \mathbf{X}_b$, and $\mathbf{Y}$.

We first describe the single-threaded version of these three steps, referred to as Seq-PANE. The multi-threaded version that boosts efficiency further is elaborated later.

Step 1. Forward and backward affinity approximation. In order to avoid numerous random walks to compute exact node-attribute values, we transform forward and backward affinity in Equations (2) and (3) into their matrix forms and utilize iterative matrix multiplications to efficiently approximate forward and backward affinity matrices with error

[2] In the degenerate case that v_l is not associated with any attribute, *e.g.*, v_1 in Figure 1(ii), we simply restart the random walk from the source node v_i, and repeat the process.

[3] The PMI quantifies how much more- or less likely we are to see the two events co-occur, given their individual probabilities, and relative to the case where they are completely independent.

guarantee and in linear time complexity, without actually sampling random walks.

Observe that in Equations (2) and (3), the key for forward and backward affinity computation is to obtain $p_f(v_i, r_j)$ and $p_b(v_i, r_j)$ for every pair $(v_i, r_j) \in V \times R$. Recall that $p_f(v_i, r_j)$ is the probability that a forward random walk starting from node v_i picks attribute r_j, while $p_b(v_i, r_j)$ is the probability of a backward random walk from attribute r_j stopping at node v_i. Given nodes v_i and v_l, denote $\pi(v_i, v_l)$ as the probability that a random walk starting from v_i stops at v_l, *i.e.*, the random walk score of v_l with respect to v_i. By definition, $p_f(v_i, r_j) = \sum_{v_l \in V} \pi(v_i, v_l) \cdot \mathbf{R}_r[v_l, r_j]$, where $\mathbf{R}_r[v_l, r_j]$ is the probability that node v_l picks attribute r_j, according to Equation (1). Similarly, $p_b(v_i, r_j)$ is formulated as $p_b(v_i, r_j) = \sum_{v_l \in V} \mathbf{R}_c[v_l, r_j] \cdot \pi(v_l, v_i)$, where $\mathbf{R}_c[v_l, r_j]$ is the probability that attribute r_j picks node v_l from all nodes having r_j based on their attribute weights. By the definition of random walk scores in [7], we can derive the matrix form of p_f and p_b as follows.

$$\begin{aligned} \mathbf{P}_f &= \alpha \textstyle\sum_{\ell=0}^{\infty} (1-\alpha)^\ell \mathbf{P}^\ell \cdot \mathbf{R}_r, \\ \mathbf{P}_b &= \alpha \textstyle\sum_{\ell=0}^{\infty} (1-\alpha)^\ell \mathbf{P}^{\top \ell} \cdot \mathbf{R}_c, \end{aligned} \tag{5}$$

where $\mathbf{P}$ is the random walk matrix (a.k.a transition matrix) of G and $\mathbf{P}^\ell[v_i, v_j]$ denotes the probability that a length-ℓ ($\ell \geq 1$) random walk from node v_i would end at node v_j. We only consider t iterations to approximate $\mathbf{P}_f$ and $\mathbf{P}_b$ in Equation (6), where t is set to $\frac{\log(\epsilon)}{\log(1-\alpha)} - 1$ and ϵ is an additive error threshold. This ensures $|\mathbf{P}_f[v_i, r_j] - \mathbf{P}_f^{(t)}[v_i, r_j]| \leq \epsilon$ and $|\mathbf{P}_b[v_i, r_j] - \mathbf{P}_b^{(t)}[v_i, r_j]| \leq \epsilon$ for every $(v_i, r_j) \in V \times R$.

$$\mathbf{P}_f^{(t)} = \alpha \sum_{\ell=0}^{t} (1-\alpha)^\ell \mathbf{P}^\ell \mathbf{R}_r, \ \mathbf{P}_b^{(t)} = \alpha \sum_{\ell=0}^{t} (1-\alpha)^\ell \mathbf{P}^{\top \ell} \mathbf{R}_c. \tag{6}$$

Then, we normalize $\mathbf{P}_f^{(t)}$ by columns and $\mathbf{P}_b^{(t)}$ by rows as follows.

$$\widehat{\mathbf{P}}_f^{(t)}[v_i, r_j] = \frac{\mathbf{P}_f^{(t)}[v_i, r_j]}{\sum_{v_l \in V} \mathbf{P}_f^{(t)}[v_l, r_j]}, \ \widehat{\mathbf{P}}_b^{(t)}[v_i, r_j] = \frac{\mathbf{P}_b^{(t)}[v_i, r_j]}{\sum_{r_l \in R} \mathbf{P}_b^{(t)}[v_i, r_l]}$$

After normalization, we compute $\mathbf{F}'$ and $\mathbf{B}'$ according to the definitions of forward and backward affinity as follows.

$$\mathbf{F}' = \log(n \cdot \widehat{\mathbf{P}}_f^{(t)} + 1), \ \mathbf{B}' = \log(d \cdot \widehat{\mathbf{P}}_b^{(t)} + 1). \tag{7}$$

In order to obtain the embedding vectors of all nodes and attributes, *i.e.*, $\mathbf{X}_f, \mathbf{X}_b$, and $\mathbf{Y}$, we need to jointly factorize the approximate forward and backward affinity matrices $\mathbf{F}'$ and $\mathbf{B}'$. This can be done based on the *cyclic coordinate descent* (*CCD*) framework, which iteratively updates each embedding value towards optimizing the objective function in Equation (4). However, a direct application of CCD, starting from random initial values of the embeddings, requires numerous iterations to converge, leading to prohibitive overheads. Furthermore, CCD computation itself is expensive, especially on large-scale graphs. To overcome these challenges, we firstly propose a greedy initialization method to facilitate fast convergence (Step 2), and then design efficient techniques to refine the initial embeddings, including dynamic maintenance and partial updates of intermediate results to avoid redundant computations in CCD (Step 3), ideas that are inspired from data management techniques.

Step 2. Greedy initialization of the embeddings. In many optimization problems, all we need for efficiency is a good initialization. Thus, a key component in the joint factorization is such an initialization of embedding values, based on *singular value decomposition* (*SVD*). Note that unlike other matrix factorization problems, here SVD cannot be directly utilized to solve our problem because the objective function in Equation (4) requires the joint factorization of both the forward and backward affinity matrices at the same time.

Specifically, the initialization method first employs an efficient randomized SVD algorithm [10] to decompose $\mathbf{F}'$ into $\mathbf{U} \in \mathbb{R}^{n \times \frac{k}{2}}, \mathbf{\Sigma} \in \mathbb{R}^{\frac{k}{2} \times \frac{k}{2}}, \mathbf{V} \in \mathbb{R}^{d \times \frac{k}{2}}$, and then initializes $\mathbf{X}_f = \mathbf{U}\mathbf{\Sigma}$ and $\mathbf{Y} = \mathbf{V}$, which satisfies $\mathbf{X}_f \cdot \mathbf{Y}^\top \approx \mathbf{F}'$. In other words, this initialization immediately gains a good approximation of the forward affinity matrix.

Recall that our objective function in Equation (4) also aims to find $\mathbf{X}_b$ such that $\mathbf{X}_b \mathbf{Y}^\top \approx \mathbf{B}'$, *i.e.*, to approximate the backward affinity matrix well. We observe that matrix $\mathbf{V}$ (*i.e.*, $\mathbf{Y}$) returned by exact SVD is *unitary*, *i.e.*, $\mathbf{Y}^\top \mathbf{Y} = \mathbf{I}$, which implies that $\mathbf{X}_b \approx \mathbf{X}_b \mathbf{Y}^\top \mathbf{Y} \approx \mathbf{B}'\mathbf{Y}$. Accordingly, we seed $\mathbf{X}_b$ with $\mathbf{B}'\mathbf{Y}$. This initialization of $\mathbf{X}_b$ also leads to a relatively good approximation of the backward affinity matrix. Consequently, the number of iterations required by CCD is drastically reduced (shown in Section 4).

Step 3. Efficient refinement of the initial embeddings. After initializing $\mathbf{X}_f, \mathbf{X}_b$ and $\mathbf{Y}$, we apply CCD to refine the embedding vectors according to our objective function in Equation (4). The basic idea of CCD is to cyclically iterate through all entries in $\mathbf{X}_f, \mathbf{X}_b$ and $\mathbf{Y}$, minimizing the objective function with respect to each entry (*i.e.*, coordinate direction). Specifically, in each iteration, CCD updates each entry of $\mathbf{X}_f, \mathbf{X}_b$ and $\mathbf{Y}$ according to the following rules:

$$\mathbf{X}_f[v_i, l] \leftarrow \mathbf{X}_f[v_i, l] - \mu_f(v_i, l), \tag{8}$$

$$\mathbf{X}_b[v_i, l] \leftarrow \mathbf{X}_b[v_i, l] - \mu_b(v_i, l), \tag{9}$$

$$\mathbf{Y}[r_j, l] \leftarrow \mathbf{Y}[r_j, l] - \mu_y(r_j, l), \tag{10}$$

with $\mu_f(v_i, l), \mu_b(v_i, l)$ and $\mu_y(r_j, l)$ computed by:

$$\mu_f(v_i, l) = \frac{\mathbf{S}_f[v_i] \cdot \mathbf{Y}[:, l]}{\mathbf{Y}^\top[l] \cdot \mathbf{Y}[:, l]}, \quad \mu_b(v_i, l) = \frac{\mathbf{S}_b[v_i] \cdot \mathbf{Y}[:, l]}{\mathbf{Y}^\top[l] \cdot \mathbf{Y}[:, l]}, \tag{11}$$

$$\mu_y(r_j, l) = \frac{\mathbf{X}_f^\top[l] \cdot \mathbf{S}_f[:, r_j] + \mathbf{X}_b^\top[l] \cdot \mathbf{S}_b[:, r_j]}{\mathbf{X}_f^\top[l] \cdot \mathbf{X}_f[:, l] + \mathbf{X}_b^\top[l] \cdot \mathbf{X}_b[:, l]}, \tag{12}$$

where $\mathbf{S}_f = \mathbf{X}_f \mathbf{Y}^\top - \mathbf{F}'$ and $\mathbf{S}_b = \mathbf{X}_b \mathbf{Y}^\top - \mathbf{B}'$.

However, directly applying the above updating rules to learn $\mathbf{X}_f, \mathbf{X}_b$, and $\mathbf{Y}$ is inefficient, leading to numerous redundant matrix operations. Hence, in each iteration of CCD, we first fix $\mathbf{Y}$ and updates each row of $\mathbf{X}_f$ and $\mathbf{X}_b$, and then updates each column of $\mathbf{Y}$ with $\mathbf{X}_f$ and $\mathbf{X}_b$ fixed. According to Equations (11) and (12), $\mu_f(v_i, l)$, $\mu_b(v_i, l)$, and $\mu_y(r_j, l)$ are pertinent to $\mathbf{S}_f[v_i]$, $\mathbf{S}_b[v_i]$, and $\mathbf{S}_f[:, r_j], \mathbf{S}_b[:, r_j]$ respectively, where $\mathbf{S}_f$ and $\mathbf{S}_b$ further depend on embedding vectors $\mathbf{X}_f$, $\mathbf{X}_b$ and $\mathbf{Y}$. Therefore, whenever $\mathbf{X}_f[v_i, l], \mathbf{X}_b[v_i, l]$, and $\mathbf{Y}[r_j, l]$ are updated in the iteration, $\mathbf{S}_f$ and $\mathbf{S}_b$ need to be updated accordingly.

Directly recomputing $\mathbf{S}_f$ and $\mathbf{S}_b$ by $\mathbf{S}_f = \mathbf{X}_f \mathbf{Y}^\top - \mathbf{F}'$ and $\mathbf{S}_b = \mathbf{X}_b \mathbf{Y}^\top - \mathbf{B}'$ whenever an entry in $\mathbf{X}_f, \mathbf{X}_b$ and, $\mathbf{Y}$ is updated is expensive. Instead, we dynamically maintain and partially update $\mathbf{S}_f$ and $\mathbf{S}_b$ according to Equations (13)-(15). Specifically, when $\mathbf{X}_f[v_i, l]$ and $\mathbf{X}_b[v_i, l]$ are updated, we update $\mathbf{S}_f[v_i]$ and $\mathbf{S}_b[v_i]$ respectively in $O(d)$ time by

$$\mathbf{S}_f[v_i] \leftarrow \mathbf{S}_f[v_i] - \mu_f(v_i, l) \cdot \mathbf{Y}[:, l]^\top. \tag{13}$$

$$\mathbf{S}_b[v_i] \leftarrow \mathbf{S}_b[v_i] - \mu_b(v_i, l) \cdot \mathbf{Y}[:, l]^\top. \tag{14}$$

Whenever $\mathbf{Y}[r_j, l]$ is updated, both $\mathbf{S}_f[:, r_j]$ and $\mathbf{S}_b[:, r_j]$ are updated in $O(n)$ time by

$$\begin{aligned} \mathbf{S}_f[:, r_j] &\leftarrow \mathbf{S}_f[:, r_j] - \mu_y(r_j, l) \cdot \mathbf{X}_f[:, l], \\ \mathbf{S}_b[:, r_j] &\leftarrow \mathbf{S}_b[:, r_j] - \mu_y(r_j, l) \cdot \mathbf{X}_b[:, l]. \end{aligned} \tag{15}$$

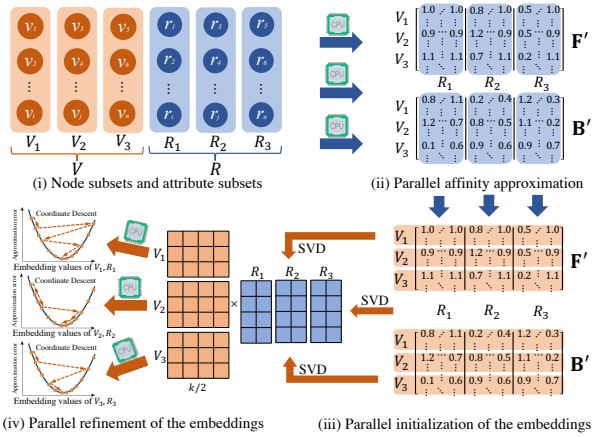


Figure 2: Overview of Par-PANE.

Complexity Analysis. Step 1 runs in $O(mdt) = O\left(md \cdot \log \frac{1}{\epsilon}\right)$ time. Meanwhile, given $\mathbf{F}' \in \mathbb{R}^{n \times d}$ as input, randomized SVD in Step 2 requires $O(ndkt)$ time [10]. The computation of $\mathbf{S}_f, \mathbf{S}_b$ in Step 3 costs $O(ndk)$ time. In addition, the t iterations of CCD for updating $\mathbf{X}_f, \mathbf{X}_b$ and $\mathbf{Y}$ take $O(ndkt) = O(ndk \log \frac{1}{\epsilon})$ time. Therefore, the overall time complexity of Seq-PANE is $O\left((md + ndk) \cdot \log\left(\frac{1}{\epsilon}\right)\right)$.

3.4 Par-PANE: A Parallel Algorithm

Although Seq-PANE runs in linear time to the size of the input network, as we shall see in Section 4, it still consumes substantial amount of time in handling massive attributed networks in practice. We now present a parallel version of PANE, called Par-PANE, that draws inspiration from the exploitation of multi-core CPUs in many modern data management techniques and beyond to enhance performance.

It is challenging to develop a parallel algorithm achieving linear scalability to the number of threads on a multi-core CPU. PANE involves intensive matrix computation, factorization, and CCD updates, which are non-trivial to parallelize. Maintenance of the intermediate result of each thread and combining them to create the final result further aggrandize this challenge. We address these challenges in Par-PANE.

Figure 2 depicts an overview of Par-PANE. Compared to the Seq-PANE, Par-PANE takes as input an additional parameter, the number of threads n_b, and randomly partitions the node set V, as well as the attribute set R, into n_b subsets with equal size, denoted as $\mathcal{V}$ and $\mathcal{R}$, respectively. It parallelizes the three key steps in Seq-PANE as follows.

Step 1. Parallel forward and backward affinity approximation. We adopt block matrix multiplication to to estimate forward and backward affinity matrices $\mathbf{F}'$ and $\mathbf{B}'$ in a parallel manner. After obtaining $\mathbf{R}_r$ and $\mathbf{R}_c$ based on Equation (1), we divide $\mathbf{R}_r$ and $\mathbf{R}_c$ into matrix blocks according to two input parameters, the node subsets $\mathcal{V} = \{V_1, V_2, \cdots, V_{n_b}\}$ and attribute subsets $\mathcal{R} = \{R_1, R_2, \cdots, R_{n_b}\}$. Then, the matrix multiplications for computing $\mathbf{P}_f^{(t)}$ and $\mathbf{P}_b^{(t)}$ are parallelized, using n_b threads in t iterations. Then, n_b matrix blocks $\mathbf{P}_{f_i}{}^{(t)}$ (resp. $\mathbf{P}_{b_i}{}^{(t)}$) are concatenated horizontally together as $\mathbf{P}_f{}^{(t)}$ (resp. $\mathbf{P}_b{}^{(t)}$) in the main thread. Afterwards, we normalize $\widehat{\mathbf{P}}_f^{(t)}$ and $\widehat{\mathbf{P}}_b^{(t)}$, and then start n_b threads to compute $\mathbf{F}'$ and $\mathbf{B}'$ block by block in parallel, based on the definitions of forward and backward affinity.

Table 1: Datasets. (K=10^3, M=10^6)

Name	$\lvert V\rvert$	$\lvert E_V\rvert$	$\lvert R\rvert$	$\lvert E_R\rvert$	$\lvert L\rvert$	Refs
Citeseer	3.3K	4.7K	3.7K	105.2K	6	[8, 9, 11, 14, 16, 19]
Facebook	4K	88.2K	1.3K	33.3K	193	[9]
TWeibo	2.3M	50.7M	1.7K	16.8M	8	-
MAG	59.3M	978.2M	2K	434.4M	100	-

Step 2. Parallel initialization of the embeddings. To further improve the efficiency of the joint affinity matrix factorization process, we design a parallel algorithm with a split-and-merge-based parallel SVD technique for embedding vector initialization. It takes as input $\mathbf{F}'$, $\mathbf{B}'$, $\mathcal{V}$, and k. Based on $\mathcal{V}$, it splits matrix $\mathbf{F}'$ into n_b blocks and launches n_b threads. Then, the i-th thread applies RandSVD to block $\mathbf{F}'[V_i]$ generated by the rows of $\mathbf{F}'$ based on node set $V_i \in \mathcal{V}$. After obtaining $\mathbf{V}_1, \cdots, \mathbf{V}_{n_b}$, we merge these matrices by concatenating $\mathbf{V}_1, \cdots, \mathbf{V}_{n_b}$ into $\mathbf{V} = [\mathbf{V}_1 \ \cdots \ \mathbf{V}_{n_b}]^\top \in \mathbb{R}^{\frac{kn_b}{2} \times d}$, and then applies RandSVD over it to obtain $\mathbf{W} \in \mathbb{R}^{\frac{kn_b}{2} \times \frac{k}{2}}$ and $\mathbf{Y} \in \mathbb{R}^{d \times \frac{k}{2}}$. Next, it creates n_b threads, and uses the i-th thread to handle node subset V_i for initializing embedding vectors $\mathbf{X}_f[V_i]$ and $\mathbf{X}_b[V_i]$, as well as computing intermediate results $\mathbf{S}_f$ and $\mathbf{S}_b$ for CCD.

Step 3. Parallel refinement of the initial embeddings. After obtaining initial embeddings $\mathbf{X}_f, \mathbf{X}_b$, and $\mathbf{Y}$, we train them by CCD in parallel based on subsets $\mathcal{V}$ and $\mathcal{R}$, in t iterations. In each iteration, we first fix $\mathbf{Y}$ and launches n_b threads to update $\mathbf{X}_f$ and $\mathbf{X}_b$ in parallel by blocks according to $\mathcal{V}$, and then updates $\mathbf{Y}$ using the n_b threads in parallel by blocks according to $\mathcal{R}$, with $\mathbf{X}_f$ and $\mathbf{X}_b$ fixed.

Note that Par-PANE does not return exactly the same outputs as Seq-PANE, as some modules (*e.g.*, the parallel version of SVD) introduce additional error. Nevertheless, as demonstrated in Section 4, the degradation of result quality in Par-PANE is small, but the speedup is significant.

Complexity Analysis. Based on the complexity analysis of Seq-PANE, it can be shown that the computational time complexity per thread in Par-PANE is $O\left(\frac{md+ndk}{n_b} \cdot \log\left(\frac{1}{\epsilon}\right)\right)$.

4. EXPERIMENTS

In this section, we investigate the performance of PANE on two tasks (link prediction and node classification). All experiments are conducted on a Linux machine powered by an Intel Xeon(R) E7-8880 v4@2.20GHz CPUs and 1TB RAM. The codes of all algorithms are collected from their respective authors, and all are implemented in Python. The code of PANE is available at https://github.com/AnryYang/PANE.

Datasets. Table 1 lists the datasets (available at https://pytorch-geometric.readthedocs.io/en/latest/modules/datasets.html) used in our experiments. $|V|$ and $|E_V|$ denote the number of nodes and edges in the graph, whereas $|R|$ and $|E_R|$ represent the number of attributes and node-attribute associations (*i.e.*, nonzero entries in attribute matrix $\mathbf{R}$), respectively. In addition, L is the set of *node labels* used in the node classification task. *Citeseer* and *Facebook* are benchmark datasets used in prior work. *TWeibo* is extracted from *Tencent Weibo* social network. *MAG* is extracted from the *Microsoft Academic Knowledge Graph*.

4.1 Experiments Setup

Baselines and Parameter Settings. We compare Seq-PANE and Par-PANE against 10 state-of-the-art competitors: eight

Table 2: Link prediction performance.

Method	*Citeseer*		*Facebook*		*TWeibo*		*MAG*	
	AUC	AP	AUC	AP	AUC	AP	AUC	AP
NRP	0.86	0.808	0.969	0.973	0.967	0.979	0.915	0.92
GATNE	0.687	0.767	0.961	0.954	-	-	-	-
TADW	0.895	0.868	0.752	0.793	-	-	-	-
BANE	0.899	0.873	0.796	0.795	-	-	-	-
PRRE	0.895	0.855	0.899	0.884	-	-	-	-
STNE	0.71	0.781	0.962	0.957	-	-	-	-
CAN	0.734	0.652	0.714	0.639	-	-	-	-
LQANR	0.916	0.916	0.951	0.917	-	-	-	-
Seq-PANE	0.932	0.919	0.982	0.982	0.976	0.986	0.96	0.965
Par-PANE	0.929	0.916	0.98	0.979	0.975	0.985	0.958	0.962

recent ANE methods including BANE [16], CAN [9], STNE [8], PRRE [19], TADW [14], ARGA [11], DGI [13] and LQANR [15], one state-of-the-art homogeneous network embedding method NRP [17], and one latest attributed heterogeneous network embedding algorithm GATNE [2].

The parameters of all competitors are set as suggested in their respective papers. For Seq-PANE and Par-PANE, by default we set error threshold $\epsilon = 0.015$ and random walk stopping probability $\alpha = 0.5$. We use $n_b = 10$ threads for Par-PANE. Unless otherwise specified, we set space budget $k = 128$. In our study, a method is excluded if it cannot finish training within one week.

Performance metrics. Following [9, 11], we adopt the *Area Under Curve* (AUC) and *Average Precision* (AP) metrics to measure the performance of the methods for the link prediction task. We use *Micro-F1* to measure node classification performance [9,15]. Lastly, we use running time to measure efficiency and scalability.

4.2 Effectiveness

Link Prediction. Link prediction aims to predict the edges that are most likely to form between nodes. We first randomly remove 30% edges in input graph G, obtaining a residual graph G' and a set of the removed edges. We then randomly sample the same amount of non-existing edges as negative edges. The test set E' contains both the removed edges and the negative edges. We run PANE and all competitors on the residual graph G' to produce embedding vectors, and then evaluate the link prediction performance with E' as follows. For PANE, we calculate $p(v_i, v_j)$ as the prediction score of the directed edge (v_i, v_j):

$$p(v_i, v_j) = \sum_{r_l \in R} (\mathbf{X}_f[v_i] \cdot \mathbf{Y}[r_l]^\top) \cdot (\mathbf{X}_b[v_j] \cdot \mathbf{Y}[r_l]^\top) \approx \sum_{r_l \in R} \mathbf{F}[v_i, r_l] \cdot \mathbf{B}[v_j, r_l]. \quad (16)$$

We adopt four prediction methods (inner product, cosine similarity, Hamming distance, and edge features) over each method and report the best performance on each dataset.

Table 2 reports the AUC and AP scores of representative methods on each dataset. Observe that these scores for Seq-PANE are similar or superior to the competitors over all datasets. Furthermore, Par-PANE has comparable performance with Seq-PANE over all datasets.

Node Classification. Node classification predicts the node labels. We first run the methods on the input attributed network G to obtain their embeddings. Then we randomly sample a certain number of nodes (ranging from 10% to 90%) to train a linear support-vector machine (SVM) classifier and use the rest for testing. NRP, Seq-PANE, and Par-PANE generate a forward embedding vector $\mathbf{X}_f[v_i]$ and a backward embedding vector $\mathbf{X}_b[v_i]$ for each node $v_i \in V$. So we normalize the forward and backward embeddings of each node v_i, and then concatenate them as the feature representation of v_i to be fed into the classifier. We repeat for 5 times and report the average performance.

Figure 3 depicts the Micro-F1 results when varying the percentage of nodes used for training from 10% to 90% (*i.e.*, 0.1 to 0.9). Both versions of PANE consistently outperform all competitors on all datasets, demonstrating its effectiveness in capturing the topology and attribute information of the input attributed networks. Specifically, compared with the competitors, Seq-PANE achieves a significant gain up to 17.2% on *MAG*. Over all datasets, Par-PANE has similar performance to that of Seq-PANE.

4.3 Efficiency and Scalability

Figure 4 reports the running times (in log-scale). It does not include the time for loading datasets and outputting embeddings. PANE is significantly faster than all ANE competitors, often by orders of magnitude. Specifically, on *TWeibo* and *MAG*, most existing ANE solutions cannot finish within a week, while our proposed solutions are able to handle them efficiently. Observe that Par-PANE is up to 9 times faster than Seq-PANE over all datasets. For instance, on *MAG* dataset, Par-PANE requires 11.9 hours while Seq-PANE takes about five days, emphasizing the benefits brought by our parallelization techniques in Section 3.4. Importantly, this is achieved without compromising on result quality.

Figure 5a depicts the speedup of Par-PANE over single-thread version on *TWeibo* when varying the number of threads n_b from 1 to 20. When n_b increases, Par-PANE becomes faster than single-thread PANE, demonstrating the parallel scalability of Par-PANE. Figures 5b and 5c report the running time of Par-PANE when varying space budget k and error threshold ϵ, respectively. In Figure 5b, observe that the running time remain stable and grows slowly with increasing k. In Figure 5c, the running time of Par-PANE decreases considerably with increasing ϵ, which is consistent with our analysis that PANE runs in linear to $\log(1/\epsilon)$.

5. FUTURE WORK

PANE opens up future research in multiple directions. First, state-of-the-art ANE frameworks do not provide any explanation of results of various downstream tasks such as link prediction. While a lack of explanation may not be critical for some applications (*e.g.*, friend recommendation in a social network), in others it is paramount for the acceptability and usage of an ANE framework. For instance, a biological signaling network models interactions (*i.e.*, biochemical reactions) between molecular species in a biological system. An example network is the human cancer signaling network [6] (CSN), which can be modeled as an attributed network. Predicting links between molecules (*e.g.*, proteins, genes) in a CSN without justification is unsatisfactory, as an oncologist would like to know the biological reasons behind such prediction. PANE could be the basis for a framework to build such explanation capabilities at a low cost.

Second, PANE makes remarkable progress in efficiency and scalability on CPUs. Naturally, further inroads can be made on performance by exploiting a GPU/multi-GPU environment. Further, while networks such as CSN may not evolve rapidly, many other real-world networks do (*e.g.*, social networks). These networks may also have different types of

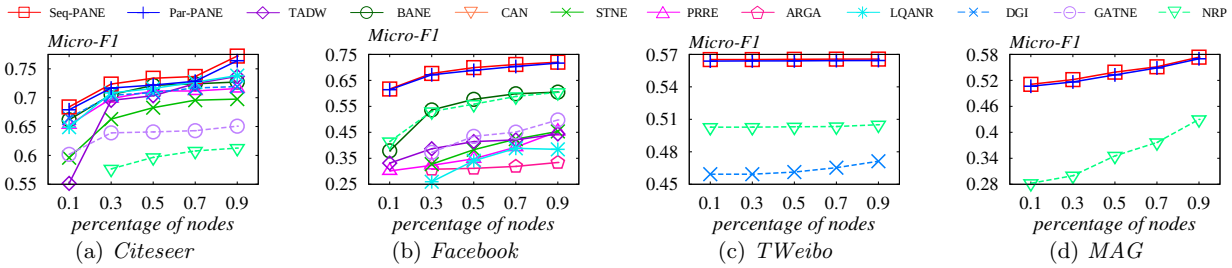


Figure 3: Node classification results (best viewed in color).

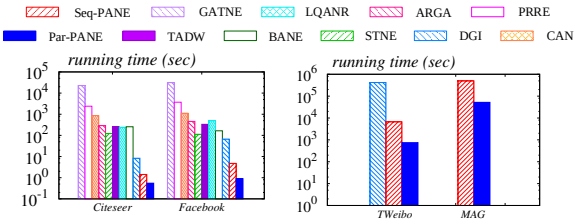


Figure 4: Running time (best viewed in color).

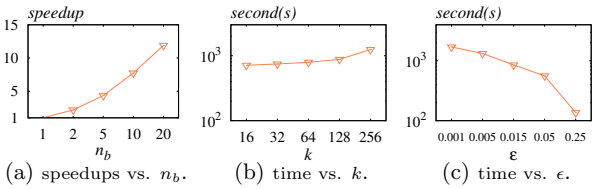


Figure 5: Efficiency with varying parameters on *TWeibo*.

links. Hence, it is also important to explore how PANE can be expanded to handle dynamic and heterogeneous networks.

Last but not the least, PANE can be a substrate to build scalable ANE-based solutions for real-world applications.For example, consider the problem of *target combination prediction* in signaling networks [3], where the goal is to predict target (nodes) combinations that can effectively modulate a set of *disease nodes*[4] in order to achieve a specific therapeutic goal (*e.g.*, reducing the activity of ERKPP protein by 50%). An *in silico* solution to this problem can aid in early rejection of unsuitable targets and guide the design of further *in vitro* and *in vivo* drug combination experiments, thereby reducing the cost and time for drug development. An effective solution needs to analyze the topology of the neighborhood of diseases nodes along with their disease-specific roles in order to capture crosstalks between pathways and their impact on targets. We are currently exploring how PANE can facilitate this by analyzing topological and disease-related attribute similarities of the neighborhood nodes encoded in their embeddings.

6. ACKNOWLEDGMENTS

This work is supported by the National University of Singapore SUG grant R-252-000-686-133, Singapore Government AcRF Tier-2 Grant MOE2019-T2-1-029, NPRP grant #NPRP10-0208-170408 from the Qatar National Research Fund (Qatar Foundation), and the financial support (1-BE3T) of research project (P0033898) from the Hong Kong Polytechnic University. The findings herein reflect the work, and are solely the responsibility, of the authors.

[4]A disease node is a molecule that is either involved in some dysregulated biological processes implicated in a disease (*e.g.*, breast cancer) or is of interest due to its potential role (*e.g.*, oncogene) in the disease.

7. REFERENCES

[1] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using BANKS. In *ICDE*, pages 431–440, 2002.

[2] Y. Cen, X. Zou, J. Zhang, H. Yang, J. Zhou, and J. Tang. Representation learning for attributed multiplex heterogeneous network. In *KDD*, pages 1358–1368, 2019.

[3] H. E. Chua, S. S. Bhowmick, and L. Tucker-Kellogg. Synergistic target combination prediction from curated signaling networks: Machine learning meets systems biology and pharmacology. *Methods*, 129:60–80, 2017.

[4] K. W. Church and P. Hanks. Word association norms, mutual information, and lexicography. *Computational linguistics*, 16(1):22–29, 1990.

[5] P. Cui, X. Wang, J. Pei, and W. Zhu. A survey on network embedding. *TKDE*, 31(5):833–852, 2019.

[6] Q. Cui, Y. Ma, M. Jaramillo, H. Bari, A. Awan, S. Yang, S. Zhang, L. Liu, M. Lu, M. O'Connor-McCourt, et al. A map of human cancer signaling. *Molecular systems biology*, 3(1):152, 2007.

[7] Y. Fujiwara, M. Nakatsuji, M. Onizuka, and M. Kitsuregawa. Fast and exact top-k search for random walk with restart. *PVLDB*, 5(5):442–453, 2012.

[8] J. Liu, Z. He, L. Wei, and Y. Huang. Content to node: Self-translation network embedding. In *KDD*, pages 1794–1802, 2018.

[9] Z. Meng, S. Liang, H. Bao, and X. Zhang. Co-embedding attributed networks. In *WSDM*, pages 393–401, 2019.

[10] C. Musco and C. Musco. Randomized block krylov methods for stronger and faster approximate singular value decomposition. In *NeurIPS*, pages 1396–1404, 2015.

[11] S. Pan, R. Hu, G. Long, J. Jiang, L. Yao, and C. Zhang. Adversarially regularized graph autoencoder for graph embedding. In *IJCAI*, pages 2609–2615, 2018.

[12] S. Sahu, A. Mhedhbi, S. Salihoglu, J. Lin, and M. T. Özsu. The ubiquity of large graphs and surprising challenges of graph processing. *PVLDB*, 11(4):420–431, 2017.

[13] P. Veličković, W. Fedus, W. L. Hamilton, P. Liò, Y. Bengio, and R. D. Hjelm. Deep Graph Infomax. In *ICLR*, pages 1843–1852, 2019.

[14] C. Yang, Z. Liu, D. Zhao, M. Sun, and E. Chang. Network representation learning with rich text information. In *IJCAI*, pages 2111–2117, 2015.

[15] H. Yang, S. Pan, L. Chen, C. Zhou, and P. Zhang. Low-bit quantization for attributed network representation learning. In *IJCAI*, pages 4047–4053, 2019.

[16] H. Yang, S. Pan, P. Zhang, L. Chen, D. Lian, and C. Zhang. Binarized attributed network embedding. In *ICDM*, pages 1476–1481, 2018.

[17] R. Yang, J. Shi, X. Xiao, Y. Yang, and S. S. Bhowmick. Homogeneous network embedding for massive graphs via reweighted personalized pagerank. *PVLDB*, 13(5):670–683, 2020.

[18] Y. Yang. An evaluation of statistical approaches to text categorization. *Information Retrieval*, 1(1):69–90, 1999.

[19] S. Zhou, H. Yang, X. Wang, J. Bu, M. Ester, P. Yu, J. Zhang, and C. Wang. Prre: Personalized relation ranking embedding for attributed networks. In *CIKM*, pages 823–832, 2018.

Technical Perspective: Bipartite Matching: What to do in the Real World When Computing Assignment Costs Dominates Finding the Optimal Assignment

Nikos Mamoulis
University of Ioannina, Greece
nikos@cs.uoi.gr

The optimal assignment problem is a classic combinatorial optimization problem. Given a set of n agents A, a set T of m tasks, and an $n \times m$ *cost matrix* $\mathbf{C}$, the objective is to find the *matching* between A and T, which minimizes or maximizes an aggregate cost of the assigned agent-task pairs. In its standard definition, $n = m$ and we are looking for the 1-to-1 matching with the minimum total cost. From a graph theory perspective, this is a weighted bipartite graph matching problem. A classic algorithm for solving the assignment problem is the Hungarian algorithm (a.k.a. Kuhn–Munkres algorithm) [3], which bears a $O(n^3)$ computational cost (assuming that $n = m$); this is the best run-time of any strongly polynomial algorithm for this problem. There are many variants of the assignment problem, which differ in the optimization objective (i.e., minimize/maximize an aggregate cost, achieve a stable matching, maximize the number of agents matched which their top preferences, etc.) and in whether there are constraints on the number of matches for each agent or task.

Assignment problems have gained interest recently, due to the advent of applications, such as ride-hailing services, that demand fast assignment decisions. For example, given a set of available drivers and a set of passengers that request a ride, the objective is to find the matching of drivers to passengers that minimizes the overall wait time (or the maximum wait time of any passenger) [1]. Other examples include assigning mobile devices to wireless access points or cars to parking spaces based on spatial distance and capacity [4], and assigning reviewers to papers [2]. In most applications, it is assumed that the matrix $\mathbf{C}$ is readily available before running the algorithm, or that each element c_{ij} of $\mathbf{C}$ can be computed very fast (e.g., c_{ij} is Euclidean distance in [4]); hence, the bottleneck is in the assignment algorithm.

However, as the following paper unveils, there are problem instances, where the elements of the cost matrix are unknown and expensive to compute. This is especially true in real-world applications, such as ride-hailing services, where an assignment needs to be computed on-demand for dynamically generated agents and tasks in real-time. The service may have to assign thousands of available drivers to requesting passengers within limited time. As the paper shows, computing the shortest path distances between all pairs of drivers and passengers (i.e., matrix $\mathbf{C}$) well exceeds the cost of finding the optimal assignment using $\mathbf{C}$.

As also observed in previous work [4], it is not necessary to compute the assignment costs between all drivers and passengers (i.e., the entire $\mathbf{C}$), since the pairs in the optimal matching tend to be spatially close to each other. Based on this fact, the paper proposes an extension of the Hungarian algorithm that computes the matrix elements in order of their likelihood to be part of the optimal matching. In particular, a set of vertices, called landmarks, are selected in the road network graph and the distances from all other vertices to them are precomputed. The extended assignment algorithm uses the precomputed distances to landmarks and the triangle inequality, to compute effective lower bounds for the cost matrix elements. In addition, the authors propose two refinement rules, which determine when it is necessary to compute the exact matching cost of a pair, during the execution of the assignment algorithm.

The experimental results exhibit an impressive performance for the proposed algorithm. It can find the assignment between thousands of drivers and passengers by only computing less than 5% of the matrix elements and it can achieve a throughput of about 1500 assigned pairs every 15 seconds for the Grab ride-hailing service in the city of Singapore. The strong and generalizable results of this paper open the road for additional work on extending assignment algorithms with different objectives (e.g., stable matching), for problems that involve hard-to-compute pairing costs and need be solved in real time.

1. REFERENCES

[1] G. Gao, M. Xiao, and Z. Zhao. Optimal multi-taxi dispatch for mobile taxi-hailing systems. In *ICPP*, pages 294–303. IEEE Computer Society, 2016.

[2] N. M. Kou, L. H. U, N. Mamoulis, and Z. Gong. Weighted coverage based reviewer assignment. In *ACM SIGMOD*, pages 2031–2046. ACM, 2015.

[3] H. Kuhn. The Hungarian method for the assignment problem. *Naval research logistics quarterly*, 2(1-2):83–97, 1955.

[4] L. H. U, K. Mouratidis, M. L. Yiu, and N. Mamoulis. Optimal matching between spatial datasets under capacity constraints. *ACM Trans. Database Syst.*, 35(2):9:1–9:44, 2010.

Bipartite Matching: What to do in the Real World When Computing Assignment Costs Dominates Finding the Optimal Assignment

Tenindra Abeywickrama
Grab Holdings Inc.
tenindra.a@grab.com

Victor Liang
Grab Holdings Inc.
victor.liang@grab.com

Kian-Lee Tan
School of Computing
National University of Singapore
tankl@comp.nus.edu.sg

ABSTRACT

The Kuhn-Munkres (KM) algorithm is a classical combinatorial optimization algorithm that is widely used for minimum cost bipartite matching in many real-world applications, such as transportation. For example, a ride-hailing service may use it to find the optimal assignment of drivers to passengers to minimize the overall wait time. Typically, given two bipartite sets, this process involves computing the edge costs between all bipartite pairs and finding an optimal matching. However, existing works overlook the impact of edge cost computation on the overall running time. In reality, edge computation often significantly outweighs the computation of the optimal assignment itself, as in the case of assigning drivers to passengers which involves computation of expensive graph shortest paths. Following on from this, we also observe common real-world settings exhibit a useful property that allows us to incrementally compute edge costs only as required using an inexpensive lower-bound heuristic. This technique significantly reduces the overall cost of assignment compared to the original KM algorithm, as we demonstrate experimentally on multiple real-world data sets and workloads. Moreover, our algorithm is not limited to this domain and is potentially applicable in other settings where lower-bounding heuristics are available.

1. INTRODUCTION

The Kuhn-Munkres (KM) algorithm [13, 15], also known as the Hungarian Method, is a combinatorial optimization algorithm widely utilized to solve many real-world problems, particularly in transportation. The KM algorithm solves the *assignment problem*, also known as the *minimum-weight bipartite matching* problem, which involves finding an optimal pair-wise assignment of a set of *agents* to a set of *jobs*. Assigning an agent to a job is associated with some cost, thus the goal is to find an optimal assignment or *matching* of agent-job pairs, such that the overall cost is minimized (or maximized depending on the problem and desired outcome).

Assignment tasks are of particular importance in transportation problems, and the KM algorithm is widely used as a subroutine in many existing works [11, 8, 23, 21]. For example, it is used in ride-hailing services to optimally match drivers to passengers for maximum utilization of available vehicles. Other examples include computing mail delivery routes using Route Inspection, where minimum-weight bipartite matching is a subroutine or the order picking problem solved by using an approximate Traveling Salesman algorithm utilizing bipartite matching. The KM algorithm takes the assignment costs as input, hence these costs must be computed for each assignment task. However, we find that existing works overlook the significance of this step. Moreover, all of the aforementioned examples involve computing assignment costs based on computationally expensive graph shortest paths. For example, the cost to assign a car to a passenger is the wait time, which is commonly modeled by the travel time of the shortest path in a road network graph. As we discuss next, cost computation has significant implications for algorithm efficiency with increasingly expensive assignment cost metrics.

1.1 Motivation

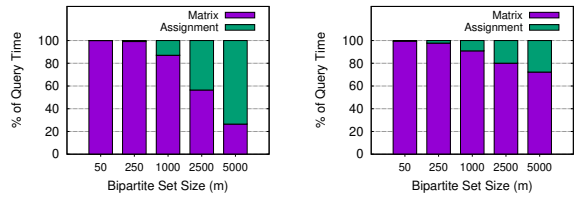
Let U be a set of agents and V be a set of jobs, both with size $m = |U| = |V|$, for which an optimal assignment is required. Also, let $c(u, v)$ be the cost of assigning agent $u \in U$ to job $v \in V$. Costs $c(u, v) \forall u \in U, v \in V$ are often conceptualized as an $m \times m$ matrix. To the best of our knowledge, all previous work utilizing KM to solve transport problems like ride-hailing assumes this matrix is provided to the KM algorithm or the cost of computing the matrix is not a bottleneck. However, in many real-world applications, computing the matrix is not only a non-trivial cost but also more computationally expensive than the assignment itself. Moreover, the matrix may need to be re-computed each time an assignment is required. Given the real-time nature of transportation problems, this may be quite frequent, which serves to only exacerbate the non-trivial cost of computing $c(u, v)$. For example, in a ride-hailing service, a new assignment is required as new cars become available and new passenger requests are received continuously in real-time. According to Fortune, popular ride-hailing services like Grab are reported to process 6 million ride requests a day, highlighting the scale of throughput required.

Our observation can be demonstrated using a simple ride-hailing assignment framework. Let us represent the cost of assigning a passenger (job) to a ride-hailing car (agent) as the travel-time of the shortest route from the car to the pas-

 This is a minor revision of the paper entitled"Optimizing Bipartite Matching in Real-World Applications by Incremental Cost Computation", published in PVLDB, Vol. 14, No. 7, 2150-8097. DOI: https://doi.org/10.14778/3450980.3450983

https://fortune.com/longform/grab-gojek-super-apps/

senger. All costs for one car (agent) can be computed by performing a single Dijkstra's single-source multiple-destination (SSMD) shortest path query. The entire cost matrix can be populated by performing m such searches. Simple worst-case analysis based on Dijkstra using Fibonacci heaps suggests that this will cost $O(m|E| + m|N| \log |N|)$ time where $|N|$ and $|E|$ are the number of vertices and edges in the road network graph and $m = |U| = |V|$. Typical real-world scenarios would see this dominate the KM algorithm time complexity of $O(m^3)$. For example, in the Singapore road network $|N|$ is over $280,000$ while m might be 100 representing finding a matching for 100 ride-hailing cars to 100 passengers. We verify this intuition in practice for the Singapore road network for varying values of m in Figure 1a using Dijkstra's search as above. As expected, the time to compute the matrix dominates the time to compute the optimal assignment for increasing m, only being overtaken when m reaches 5000. In Figure 1b we show this is still true even if a fast modern point-to-point shortest path technique like Contraction Hierarchies is used.



(a) Dijkstra (b) Contraction Hierarchies

Figure 1: Proportion of running time spent on matrix computation and finding optimal assignment on Singapore road network for varying m

1.2 Contributions

We have seen that computing the cost matrix often dominates computing the optimal assignment itself. Moreover, the cost matrix must be computed from scratch for each assignment problem and may need to be performed frequently in real-world applications such as ride-hailing. In attempting to address the scalability and throughput concerns that arise as a result, it begs the question of whether all assignment costs are even necessary to compute an optimal solution as first observed by [14]. We observe that this is also not necessarily the case due to a property exhibited by optimal assignments in typical real-world scenarios. For example, in a ride-hailing service for a particular geographic region, such as Singapore, typically passengers and drivers will be distributed in various parts of the region. It is unlikely that a driver $u \in U$ will be assigned to some passenger $v \in V$ a significant distance away. We say that such problems exhibit high *spatial locality of matching*. Using this intuition we propose a minimum-weight bipartite matching algorithm based on the KM algorithm that incrementally computes costs that are most likely to be in the optimal matching. We develop novel refinement rules using inexpensive lower-bounding heuristics to only compute costs when necessary. Notably, our technique still computes the optimal matching, but does so while computing far fewer expensive pair-wise exact assignment costs, significantly reducing the overall running time. Moreover, our technique is a drop-in replacement for the KM algorithm in any technique or framework that uses the KM as a subroutine. Our contributions can be summarized as follows:

- We identify that computing assignment costs such as graph shortest paths are more computationally expensive than finding the optimal assignment itself for workloads for real-world problems such as ride-hailing.
- We present a minimum-weight bipartite matching algorithm based on the Kuhn-Munkres algorithm that incrementally computes the exact assignment costs required for an assignment only when it is necessary according to novel pruning rules utilizing inexpensive lower-bounding heuristics.
- We implement a specialized lower-bounding heuristic for use in ride-hailing services, where the assignment cost is represented by the travel-time of the shortest path in a road network graph, adapting landmark-based lower-bounds and graph search techniques.
- Our extensive experimental investigation using large-scale real-world data sets and workloads demonstrates the significant improvement achieved by our proposed solutions with highly favorable implications for real-world scalability and throughput.

2. PRELIMINARIES

The assignment problem is often formulated as the minimum weight bipartite matching problem. Then, we are given a bipartite graph $B = (U \cup V, E_B)$ where U and V are the bipartite sets of vertices. E_B is the set of edges, and contains an edge $(u, v) \forall u \in U, v \in V$. The weight $c(u, v)$ of an edge represents the cost of assigning u to v. The assignment problem finds a *perfect matching*, where every object in U is assigned to exactly one object in V (and vice versa), such that the sum of weights over all assigned pairs is minimized. For simpler exposition, we consider equally sized sets, i.e., $m = |U| = |V|$, which in practice can be simulated by adding dummy vertices to the smaller set. Next, we describe the preliminaries for the applied setting for which our techniques are designed to be deployed.

Road Network: In the case of a ride-hailing service, the bipartite sets consist of the locations of passengers and drivers to be matched. The cost of assigning a passenger to a driver is commonly considered as the minimum travel-time for the driver to reach the passenger's location. These costs can be computed by first considering the road network $G = (V_G, E)$, where V_G is the set of vertices and E is the set of edges. Each edge $(x, y) \in E$ represents the road segments connecting junction vertices x and y with weight $w(x, y)$ representing the travel-time to traverse the edge. Note that other real positive metrics, such as physical length, can also be considered. In our context travel-time, and hence the waiting time for passengers, is most relevant. The *network distance* $d(s, t)$ between a source vertex s and destination

Note that the number of edges $|E|$ on road network graphs observes $|E| = O(|N|)$

While faster techniques for point-to-point shortest path search are available, Dijkstra is typically faster for SSMD search when many destinations are involved. This is because for increasing m the number of point-to-point queries increases by its square, explaining why the matrix computation cost percentage is still high for large values of m in Figure 1b for CH.

vertex t is the minimum sum of weights connecting vertices s and t, i.e., by the shortest path in G. Note that we consider passenger and driver locations that occur on vertices for simpler exposition and implementation, but our techniques can be extended for when this is not the case. In relation to the assignment problem, $c(u, v) = d(u, v)$.

Landmark Lower-Bounds (LLBs): Our proposed technique leverages the idea of computing an inexpensive lower-bound on the assignment cost $c(u, v)$ that is as accurate as possible. In the case of network distance as assignment cost, Landmark Lower-Bounds (LLBs) [10] are an effective lower-bound for shortest paths in graphs and can be computed cheaply. LLBs involve selecting k "landmark" vertices and computing network distances to each vertex in V from each landmark in an offline pre-processing step. During the online query phase, a lower-bound distance between any two vertices s and t may be computed using the distances to any landmark vertex l and the triangle inequality as in (1). A surprisingly accurate lower-bound can be computed by considering lower-bounds over all k landmarks as in (2), even for small values of k. Consequently, we utilize LLBs as the lower-bound on assignment cost $c(u, v)$.

$$LB_l(q, p) = |d(l, q) - d(l, p)| \leq d(q, p) \tag{1}$$

$$LB_{max}(q, p) = \max_{l \in L}(|d(l, q) - d(l, p)|) \tag{2}$$

Kuhn-Munkres Algorithm: We use the Kuhn-Munkres (KM) algorithm as the basis for our improved techniques. KM works by iteratively updating a set of labels l_u (resp. l_v) for bipartite set U (resp. V) that imply a *reduced cost* of each bipartite edge $(u, v) \in E_B$:

$$c_r(u, v) = c(u, v) - l_u - l_v \tag{3}$$

KM adjusts the labels to generate edges of zero reduced costs while maintaining the invariants below. If a *perfect matching* exists amongst these edges (referred to as the reduced graph), then this matching is the optimal solution to the minimum-weight bipartite matching problem [17].

INVARIANT 1. *The reduced cost of each edge must be non-negative, i.e.,* $c_r(u, v) \geq 0$

INVARIANT 2. *Each edge in* M *is "tight" in that it has reduced cost zero, i.e.,* $c_r(u, v) = 0$ *where* $(u, v) \in M$

KM uses augmenting paths [7] to find a perfect matching in the reduced graph. When one does not exist, the labels are adjusted by computing δ below, where $S \in U$ and $N(S) \in V$ are vertices visited by the search. We refer to [17, 7, 6] for details of these well-known techniques.

$$\delta := min\{c(u, v) - l_u - l_v : u \in S, v \notin N(S)\} \tag{4}$$

3. INCREMENTAL KUHN-MUNKRES

Recall the intuition of *spatial locality of matching*, that posits an optimal assignment for ride-hailing matching task is unlikely to assign drivers to passengers that are very far away. A simple approach to utilize this intuition might be to subdivide the region further and run the KM algorithm on each subregion separately. Naturally, this would reduce the size of m and hence the number of assignment costs that must be computed. However, this approach would no longer provide a globally optimal assignment. For example, at borders between regions, suboptimal assignment is likely to occur. In this section, we propose methods to utilize the intuition and avoid computation of exact costs, while still returning the globally optimal result.

3.1 Lower-Bounding Module

Our technique is underpinned by the ability to compute lower-bounds on edge cost $c(u, v)$ during the KM algorithm iterations. We propose an abstract Lower-Bound Module that provides the ability to compute two different lower-bounds, defined as follows:

DEFINITION 1. *(Individual Lower-Bound Edge Cost) Given vertices* $u \in U$ *and* $v \in V$*, an individual lower-bound edge-cost* $LB(u, v)$ *is a lower-bound on the true edge-cost* $c(u, v)$*, i.e.,* $LB(u, v) \leq c(u, v)$*.*

DEFINITION 2. *(Group Lower-Bound Edge Cost) Given vertex* $u \in U$*, let* $Q_u \subseteq V$ *represent the set of vertices for which the true edge cost is not known (initially* $Q_u = V$*). A group lower-bound edge cost* $LB(Q_u)$ *is a lower-bound for all edge-costs* $c(u, v) \forall v \in Q_u$*, i.e.,* $LB(Q_u) \leq c(u, v) \forall v \in Q_u$*.*

The group lower-bound edge cost is best implemented as a minimum priority queue. This allows iterative extraction of candidates from Q_u, while maintaining the definition. Moreover, the queue can be lazily updated such that the definition is met, similar to the on-demand heaps in [2]. That is, Q_u is not required to contain individual lower-bounds for all vertices in V. Next, we show how to modify the KM algorithm to use individual and group lower-bound edge costs to avoid computation of exact costs $c(u, v)$ where possible .

Note that the solution is agnostic to the implementation of Q_u and the type of cost $c(u, v)$, and can be applied to any problem setting. However, we specify the implementation for costs based on shortest paths in road network graphs where significant benefits can be gained. This is because computation of shortest paths in road network graphs is a computationally intensive task and is often used in real-world applications such as ride-hailing services and the route inspection problem. Figure 2 depicts the components of the system. The priority queues for each vertex $u \in U$ are exposed to the KM algorithm module, as is a module to compute the true cost $c(u, v)$ (when deemed necessary) using a fast shortest path technique such as G-tree [24].

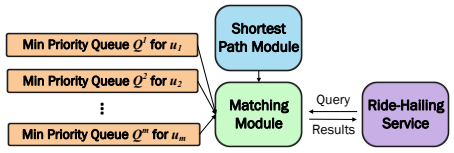


Figure 2: System Overview

3.2 Refinement Rules

We propose the Incremental Kuhn-Munkres (IKM) algorithm (Algorithm 1) that incrementally computes exact edge-costs only when necessary, utilizing the Lower-Bound Module in the process. In this section, we propose two novel

Algorithm 1 Optimized KM algorithm using refinement rules

```
1: function OptKuhnMunkres(U, V)
2:    M ← φ and initialize labels l_u = l_v = 0
3:    Initialize Lazy MPQ Q_u for each u ∈ U
4:    while M is not a perfect matching (i.e., |M|! = m) do
5:       while unmarked free u ∈ U & no augment. path do
6:          Call find-augmenting-path(u) and mark u
7:       if augmenting path P found then
8:          Augment M by P (increasing size of M by 1)
9:       else
10:         Call update-labels subroutine
11:   return minimum-weight matching M
```

refinement rules, which designate when an exact cost $c(u, v)$ must be computed during the incremental process.

Rule 1 - BFS Expansion: Let us first define *refinement* as extracting an element v from queue Q_u with the smallest individual lower-bound, and computing its cost $c(u, v)$, and then updating $LB(Q_u)$ such that Definition 2 is maintained. We can compute a lower-bound reduced cost for all vertices in Q_u based on (3), as we propose in Lemma 1.

Lemma 1. $c_r(u, v) \geq LB_r(u, v) = LB(Q_u) - l_u - l_v \forall v \in Q_u$

Proof. By Definition 2, we have $LB(Q_u) \leq c(u, v) \forall v \in Q_u$. By (3), $c_r(u, v) = c(u, v) - l_u - l_v$. Therefore, $c_r(u, v) \geq LB(Q_u) - l_u - l_v$. Substituting gives $c_r(u, v) \geq LB_r(u, v)$, thus completing the proof. □

The proof of Lemma 1 follows in a straight-forward manner given the definition of $LB(Q_u)$. During the BFS expansion in the augmenting path algorithm, the KM algorithm expands all "tight" edges, i.e., those with reduced cost zero by Invariant 2. To ensure correctness of this expansion in our algorithm, we first propose the following theorem:

Theorem 1. *Given $v \in Q_u$, if $LB_r(u, v) > 0$ where $LB_r(u, v)$ computed by the definition in Lemma 1, then edge (u, v) cannot be a tight edge.*

Proof. From Lemma 1, we have that $c_r(u, v) \geq LB_r(u, v)$. If $LB_r(u, v) > 0$, then it follows that $c_r(u, v) > 0$. Thus, $c_r(u, v) \neq 0$ and therefore edge (u, v) cannot be tight by Invariant 2. □

Theorem 1 implies the first refinement rule, which we incorporate into a modified augmenting path search as presented in Algorithm 2. If the BFS reaches vertex $v \in V$ from vertex $x \in U$ and $LB_r(x, v) \leq 0$, we iteratively refine Q_x by extracting the element in Q_x with the smallest lower-bound and updating $LB(Q_x)$ (and therefore $LB_r(x, v)$) for the vertices remaining in Q_x. This loop terminates when either (a) $LB_r(x, v) > 0$ and by Theorem 1, edge (x, v) is not tight and need not be expanded or (b) element v is extracted from Q_x. If the extracted element is not v then we save it in excess set E, which we make sure to re-insert into the queue after the loop ends, to ensure $LB(Q_x)$ remains accurate for other $v \in V$ while ensuring we only compute necessary edge costs. Note, LB_x remains correct for v even when we remove $e \neq v$ from Q_x by the definition $LB(Q_x)$.

Rule 2 - δ Computation: Exact edge costs may also be required to determine δ by (4). Let $\alpha := max(l_v) : v \notin N(S)$, i.e., the maximum label value for vertices not in set $N(S)$ defined in Section 2 (vertices in V visited by the augmenting path search). We propose an iterative process as in Algorithm 3 to refine and update δ until its final value is attained. We first propose Lemma 2 to define a lower-bound on the smallest reduced cost for any edge (u, v) where $v \in Q_u$:

Algorithm 2 Find augmenting paths given Rule 1

```
1: function find-augmenting-path(u)
2:    Initialize new queue PQ by inserting u
3:    while PQ is not empty do
4:       Extract candidate x from PQ
5:       for each neighbor v ∈ V of x do
6:          if c(x, v) unknown & LB_r(x, v) ≤ 0 by Lemma (1) then
7:             while LB_r(x, v) ≤ 0 and c(x, v) not yet computed do
8:                Extract minimum element e ∈ V from Q_x
9:                if e = v then
10:                  Compute c(x, v) and break loop
11:               else
12:                  Add e to set E and update LB_r(x, v)
13:            Re-insert all e ∈ E back to Q_x by LB(x, e)
14:         if c(x, v) was calculated and c_r(x, v) = 0 then
15:            if v is a free vertex (i..e., not covered by M) then
16:               return path P from u to v as augmenting path
17:            else
18:               Add neighbors u ∈ U of v where (u, v) ∈ M to Q
```

Lemma 2. *Let $LB_r(u) = LB(Q_u) - l_u - \alpha$. Then $LB_r(u) \leq c_r(u, v)$ for all $v \in Q_u \setminus N(S)$.*

Proof. We prove Lemma 2 by contradiction. Let us assume there exists $LB_r(u) > c_r(u, v)$ for some $v \in Q_u$. Since $c_r(u, v) = c(u, v) - l_u - l_v$ and by the definition of $LB(Q_u)$, we have $c_r(u, v) \geq LB(Q_u) - l_u - l_v$. Given our assumption and $\alpha \geq l_v$, $c_r(u, v) \geq LB(Q_u) - l_u - \alpha$. I.e., $c_r(u, v) \geq LB_r(u, v)$, contradicting our assumption. □

Now, given the definition of $LB_r(u)$ we can propose Theorem 2 to identify when to refine a Q_u.

Theorem 2. *Let $\delta_{cand} = c_r(x, y)$ be a potential δ by (4) for $x \in S, y \notin N(S)$. Given some $u \in S$, if $\delta_{cand} < LB_r(u)$, then $\delta_{cand} < c_r(u, v) \forall v \in Q_u$.*

Proof. By Lemma 2, we have $LB_r(u) \leq c_r(u, v) \forall v \in Q_u$. Therefore, if $\delta_{cand} < LB_r(u)$ then $\delta_{cand} < c_r(u, v) \forall v \in Q_u$. Thus completing the proof. □

Using Theorem 2, Algorithm 3 can iteratively refine queues until converging to the correct δ. δ_{cand} is the candidate value of δ that we will iteratively update until it is correct. We initialize δ_{cand} with the minimum reduced cost $c_r(u, v)$ amongst $u \in S$ and $v \notin N(S)$ for which $c(u, v)$ has been already calculated and infinity otherwise. Given Q_u where $u \in S$, we compute lower-bound $LB_r(u)$ using Lemma 2. While $LB_r(u) < \delta_{cand}$, we extract the minimum element from Q_u. If it is in $N(S)$ we add to an excess set E, otherwise, we try to filter it by computing an individual lower-bound using the Lower-Bounding Module according to Definition 1, thus potentially avoiding computing an expensive exact cost. Otherwise, we compute the exact cost of the edge and update δ_{cand} if it improves it. Once Q_u is sufficiently refined (i.e., $LB_r(u) \geq \delta_{cand}$), we repeat the procedure for all $u \in S$. $\delta = \delta_{cand}$ upon termination.

The incremental computation of exact costs, adjudicated by the refinement rules, ensures that no other possible δ can be lower than the one computed by Algorithm 3. Similar to the modified augmenting path search in Algorithm 2, this is

Algorithm 3 Updated labels based on Rule 2

```
1: function UPDATE-LABELS
2:   Let S ⊂ U & N(S) ⊂ V be vertices visited by FIND-AUGMENTING-PATH
3:   Set δ to min c_r(u,v) for u ∈ S and v ∉ N(S) where c(u,v) has been computed
4:   for each u ∈ S do
5:     while LB_r(u) < δ_cand with LB_r(u) by Lemma (2) do
6:       Extract minimum element e ∈ V from Q_u
7:       if e ∉ N(S) then
8:         Compute individual LB(u,e) by LB Module
9:         Set LB_r(u,e) = LB(u,e) − l_u − l_e
10:        if LB_r(u,e) < δ_cand then
11:          Compute c(u,e) and c_r(u,e)
12:          if c_r(u,e) < δ_cand then
13:            Set δ_cand = c_r(u,e)
14:        else
15:          Add e to set E and update LB_r(u)
16:      else
17:        Add e to set E and update LB_r(u)
18:    Re-insert all e ∈ E back to Q_u by LB(u,e)
19:  for each u ∈ S do
20:    Increase l_u by δ
21:  for each v ∈ N(S) do
22:    Decrease l_v by δ
```

done in a greedy heuristic way, such that we only refine (and thus compute exact costs) for edges when it is necessary. We propose Theorem 3 to show that our refinement rules still produce the same assignment as the original KM algorithm.

THEOREM 3. *The matching produced by Algorithm 1 is identical to the matching produced by the original Kuhn-Munkres algorithm using the augmenting path search method.*

Proof Sketch: To prove Theorem 3 it is sufficient to show that (a) the modified-BFS and (b) the calculated delta is the same as the original. First, (a) follows simply as Algorithm 1 applies Theorem 1 to all edges originating from $u \in U$, so no tight edges are missed during the U to V expansion. For (b), Algorithm 1 iteratively applies Theorem 2 to each $u \in S$. As such no $c_r(u,v) \forall u \in S, v \notin N(S)$ can be smaller than δ_{cand} at termination.

3.3 IKM Variants

While we proposed our techniques in a way that is agnostic to the implementations and problem setting, the efficacy of our improvement will depend highly on these factors. The accuracy of the lower-bounds (i.e., how close they are to the true edge cost) will determine how effective the filtering steps are. The net gain in performance will be determined by the overhead added by our modifications versus the time saved avoiding exact computations. We propose two variants of our IKM technique to investigate the interplay between filtering efficiency versus overhead as described below:

IKM-DIJK: In this variant, we utilize the priority queue used by Dijkstra's search from each $u \in U$ to implement Q_u. Both individual and group lower-bounds provided by the Lower-Bounding Module utilize the minimum key in the priority queue. The traditional KM implementation would simply conduct a Dijkstra search from each $u \in U$, whereas our incremental approach stops and restarts the search as necessary, potentially terminating earlier. IKM-DIJK will provide an interesting point of comparison as it essentially adds no overhead to the original KM algorithm that utilizes Dijkstra to populate the whole distance matrix.

IKM-CAG: Many road network graph query processing studies have identified the potential benefit of using off-line preprocessing to increase online query performance. As a result, many techniques to compute shortest paths, lower-bounds, and retrieve nearest objects have been proposed that utilize indexing to improve performance. For our second variant, we implement Q_u using COLT [3], which is a state-of-art-technique technique to retrieve objects by minimum lower-bounds. We utilize the ALT index [10] to provide accurate but inexpensive lower-bound computations on shortest path distances in graphs. Lastly, we utilize G-tree [24] to efficiently compute shortest path distances with a reasonable memory footprint. Both the ALT and G-tree indexes are built in an offline pre-processing step, whereas COLT is unique to the current assignment query and built online at query time. All query time overheads are included in the running times reported in all of our experiments.

Approximate KM: [18, 14] proposed an approach similar to ours in goal, from which we develop an approximate algorithm inspired by their minimum-cost flow approach and our lower-bounding heuristic. Please refer to the full paper [4] for discussion of its experimental performance.

Name	Region	# Vertices	# Edges
SIN	Singapore	289,918	632,243
E	Eastern US	3,598,623	8,708,058

Table 1: Road Network Datasets

4. EXPERIMENTS

We conduct a detailed experimental study on the performance of the Incremental Kuhn-Munkres (IKM) algorithm. First, we investigate the likely real-world impact of IKM using actual production datasets provided by Grab . Then in the second section, we study scalability and conduct sensitivity analysis using publicly available real-world datasets and carefully generated synthetic workloads. Further details of the datasets will be provided in each section, while we describe the experimental settings below.

Environment: We run experiments on a MacBook Pro running OS X (64-bit) with a 6-core Intel Core i7 2.6 CPU and 16GB memory for the production datasets, and a Ubuntu 64-bit PC with a 16-core AMD Ryzen 3700X CPU and 32GB for the public datasets. All experiments were conducted using memory-resident indexes for fast querying. We implemented all techniques in single-threaded C++ and compiled by g++ v5.4 with O3 flag, sharing subroutines and basic data structures to ensure fairness.

Techniques: We include the two variants of our IKM technique described in Section 3.3, IKM-DIJK, and IKM-GAC. We compare our techniques against variants of the traditional KM algorithm where the cost matrix is fully computed before the matching is found. These non-incremental KM variants only differ in the technique used to compute the matrix. One variant, *Dijkstra* uses a single-source multi-destination Dijkstra search from each vertex in U to populate the matrix. *G-tree* and *CH* uses point-to-point shortest path distance queries using the G-tree [24] and Contraction Hierarchies (CH) [9] indexes, respectively. The Dijkstra

https://www.grab.com/

Method	Running Time (ms)			Matrix Computations (%)			Max. Throughput (m)		
	W=15s	W=30s	W=60s	W=15s	W=30s	W=60s	W=15s	W=30s	W=60s
Dijkstra	2876ms	4595ms	9749ms	100.0%	100.0%	100.0%	m=575	m=1050	m=1675
CH	661ms	1512ms	6605ms	100.0%	100.0%	100.0%	m=575	m=800	m=1150
G-tree	280ms	599ms	2942ms	100.0%	100.0%	100.0%	m=900	m=1200	m=1650
IKM-DIJK	65ms	110ms	407ms	2.7%	3.7%	4.5%	m=1400	m=1750	m=2275
IKM-GAC	12ms	31ms	255ms	2.9%	3.5%	3.2%	m=1425	m=1775	m=2250

Table 2: Performance metrics for a real-world ride-hailing workload for the city of Singapore. Time window W is the period that ride-hailing requests are batched for which bipartite matching is then used to compute an optimal matching

and G-tree variants allow an apples-to-apples comparison of each of our improved techniques with their corresponding non-incremental counterparts. For example, the difference in running time between G-tree and IKM-GAC will show us how much efficiency is gained from fewer distance computations, while taking into account the overhead added by object retrieval and lower-bound computations.

4.1 Real-World Performance

Given the importance of the real-world applications, we evaluate techniques on real-world data sets provided by Grab for the city of Singapore in several ways as we describe next.

4.1.1 Ride-Hailing Performance

We first evaluate the performance of our techniques on a real-world ride-hailing workload for the city of Singapore.

Datasets: The dataset consists of the road network graph G for Singapore as listed in Table 1 and workload B consisting of hundreds of thousands of anonymized ride-hailing booking records completed in a 1-week period from December 2018. Each booking in set B contains the time of the booking, the driver's location, and the user's location. Both datasets are provided by Grab and originate from real-world data generated in a production setting.

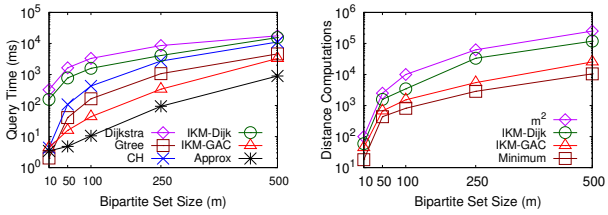
Methodology: To accurately evaluate bipartite matching performance in ride-hailing, we implement a simple batching framework based on public descriptions of real-world matching for ride-hailing applications [1]. Given a time-window W and a start time t, we select all bookings made in the time range $[t, t+W)$ from the booking set B. We then create two bipartite sets using the locations of drivers and users, respectively, from the selected bookings. We use each technique to find an optimal matching on these bipartite sets, reporting the running time and the percentage of the full cost matrix that is computed. We investigate windows W of 15, 30, and 60 seconds, and average the reported results over several randomly selected start times to reduce variability.

Running Time and Efficiency: The running times and matrix computations for each technique over all windows are listed in Table 2. The running times of our techniques, IKM-DIJK and IKM-GAC, are more than an order of magnitude less than their direct counterparts, Dijkstra and G-tree, for all values of W. The reason for this is seen in the percentage of the cost matrix that is computed by our techniques. Naturally, the original KM variants compute 100% of the cost matrix. Notably, the impressive results for IKM-GAC show that the overhead added in computing lower-bounds and retrieving objects is significantly outweighed by the time saved from reduced computations. The magnitude of improvement decreases slightly for the large window W. With a larger window, the density of driver and user locations increases, making lower-bounds less accurate. Nonetheless, the degradation is only slight, and running time is still over a magnitude better than the original KM algorithms.

Maximum Throughput: Due to the commercially-sensitive nature of the data, which is subject to non-disclosure agreements, we are not able to divulge details on the sizes of workload, particularly the average m for each window w. However, in place of this, we report the maximum throughput for each technique in Table 2. Maximum throughput is the largest possible m for which a technique can compute an optimal matching within the time window W. In real-world terms, it is the largest number of bookings that can be batched using each technique for window W, before the next batch must be computed. This is a particularly useful metric, as it will test the ability of each technique to scale to larger cities such as Jakarta and New York, which are likely to generate a far larger workload of bookings. The bipartite sets are again generated from the real-world booking set B as before, except we test increasing values of m by choosing additional bookings (in time order) until the running time is W. Table 2 shows IKM-DIJK and IKM-GAC again leads the way, reporting the highest supported throughput. Note that Dijkstra-based techniques perform relatively better here. This is because Dijkstra's running time grows linearithmically given its asymptotic complexity, while the running time of point-to-point shortest path techniques grows quadratically as it issues one query for each cell in the cost matrix. While modern techniques such as G-tree and CH have significantly improved on Dijkstra's for point-to-point shortest paths, this shows Dijkstra still works well for multi-target shortest paths.

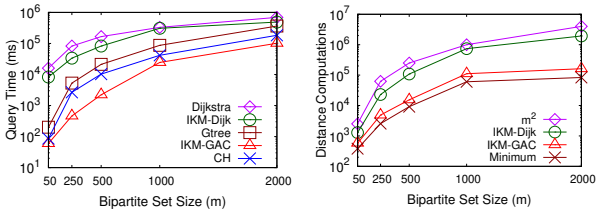
4.1.2 Sensitivity Analysis

The performance on increasing m evaluates the ability of techniques to handle increasingly large batches of ride-hailing requests. We use synthetic driver and user locations to conduct sensitivity analysis into the effect of the size of bipartite sets m. These locations are generated by selecting road network vertices uniformly at random for a given value of m. Road network vertices are more densely located in urban areas, so the coordinates of chosen vertices are more likely to be in such areas, which generally reflects booking requests. Figure 3a show that IKM-GAC improves significantly over G-tree in running time for most values of m. The exception for small values of m is due to the overhead added by IKM-GAC (such as initializing the priority queues Q_u and computing the COLT index). This overhead represents a higher proportion of running time for smaller m where the number of distance computations is small (and as such the savings are also small). In Figure 3b, we compare the number of distance computations computed by each method. Note that both Dijkstra and G-tree compute the same number of distance computations (i.e., for all pairs



(a) Running Time (b) Distance Computations

Figure 3: Singapore Dataset Performance on Varying m



(a) Running Time (b) Distance Computations

Figure 4: Eastern US Dataset Performance on Varying m

of locations), and this is represented in the m^2 line. The improvement shown on the synthetic dataset appears to be smaller than the real-world dataset. This is likely due to *spatial location of matching* being less prominent in the synthetic dataset, which we confirm experimentally. For each pair in the optimal matching, we found that objects in U were on average assigned to the 2nd to 3rd nearest object for the production dataset, experimentally confirming the presence of *spatial location of matching* in real-world datasets. On the other hand, objects were matched to increasingly further objects with increasing m (e.g., 10th nearest object for $m = 250$) for the synthetic datasets. Thus, the synthetic datasets are more challenging, and the still sizeable improvement demonstrates the robustness of our techniques. *Minimum* is an estimate on the theoretical minimum number of costs required to find the optimal matching (its derivation is outlined in [4]). IKM-GAC closeness to the minimum shows the benefit of using an accurate lower-bound [3].

4.2 Scalability Analysis

While the Singapore dataset used in the previous section provides valuable insight into the real-world performance of the techniques, we use additional publicly available datasets for further evaluation. In particular, Singapore has a relatively smaller road network and the size of the road network has a large impact on shortest path computation. Using publicly available datasets will also provide more reproducible results. To study the scalability of the techniques we study their performance on a larger road network dataset, namely, the Eastern (E) US dataset obtained from the 9th DIMACS Challenge with 3.5 million vertices. While a ride-hailing batching operation may not be performed on such a large region, road networks for big congested cities such as Jakarta have similar numbers of vertices and edges. Synthetic bipartite sets for these road networks are generated as in Section 4.1.2, however, we use larger values of m to scale with the increased road network size. We refer to the original paper [4] for experiments on more road networks.

Eastern US Dataset: Figure 4 reports the running time and number of distance computations of each technique for increasing values of m, which corresponds to having more objects to match. We largely see similar trends to the Singapore dataset. For running time (Figure 4a), the gap between IKM-GAC and IKM-DIJK is larger than for Singapore. This can be explained by the linearithmic time complexity of Dijkstra of $O(|E|+|V_G|\log|V_G|)$. Larger numbers of road network vertices $|V_G|$ for the Eastern US road network compared to Singapore (Table 1) results in more costly distance computations. Nonetheless, the improvement of IKM-DIJK is essentially free, as IKM-DIJK introduces no overhead compared to plain Dijkstra, as it uses the same priority queue as the Dijkstra search. Furthermore, the relative improvement of IKM-GAC is higher than IKM-DIJK over its orignal KM counterpart. Given that paths are more costly to compute in a bigger (or denser) road network, this shows that it is worthwhile to pre-process data offline to accelerate online shortest path queries, with fast shortest path distance techniques like G-tree scaling better with increasing size of the road network than Dijkstra. Moreover, it suggests that even the overhead added at query time by IKM-GAC (e.g., construction of the COLT index), which is included in running times reported in all figures, is worthwhile. We also verify the observation made for maximum throughput in Section 4.1.1, with running time of techniques beginning to converge with increasing m as the time to find an optimal assignment begins to dominate cost computation time.

5. RELATED WORK

Given their popularity, real-world ride-hailing apps have spawned a growing body of research. In particular, the Kuhn-Munkres (KM) algorithm is widely used as a subroutine in real-world ride-hailing systems. For example, ride-hailing service Didi reportedly uses KM in the driver dispatch framework [21]. Similarly, Uber frame driver-rider matching as a combinatorial optimization problem to minimize the overall wait time, which is typically solved by KM [1]. The assignment problem, bipartite matching, and Kuhn-Munkres are utilized in many ride-hailing and taxi studies [23, 8, 11]. Our techniques can potentially improve running time in these frameworks as a drop-in replacement for the KM algorithm. Other work has focused on improving different aspects of ride-hailing performance, such as predictive algorithms to increase the likelihood of the driver accepting the allocated job [22]. Such considerations are likely orthogonal to our work, as the cost of allocating a passenger to a driver will still incorporate travel cost.

Since the advent of the Kuhn-Munkres algorithm [13, 15], the time complexity for the general assignment problem has not been significantly improved after [6, 19] improved the original $O(n^4)$ time to $O(n^3)$. Further improvements have come primarily in the form of specialized domains such as considering bounded integer weights [16] or improvements through clever heuristics that work extremely well in practice [12]. Other approaches have attempted to find approximate solutions that trade running time for accuracy [5]. These works are complementary to our technique because they increase the relative running time of computing the cost

http://www.dis.uniroma1.it/%7Echallenge9/

matrix. For example in Figure 1, these techniques would decrease the time taken up by the assignment for increasing values of m, thus making it even more necessary to reduce computations. [4] discusses further related work.

An "incremental" variant of the assignment problem has also been proposed [20], but their definition of incremental involves updating an optimal assignment based on new objects. Some techniques [14, 18] improve the efficiency of the minimum cost flow algorithm by attempting to compute only a partial bipartite graph and terminate early. While utilizing a similar strategy to us, our techniques also attempt to terminate early by computing a partial bipartite graph *and* attempt do so while only computing lower-bounds on the edges we do compute, wherever possible. This is necessary in our problem domain as, for example, road network shortest paths are significantly more expensive to compute than the Euclidean distance costs in [14]. However, it suggests a possible future avenue for research in that we may be able to also optimize the running time of the assignment, which would be helpful when that running time exceeds the cost matrix computation, e.g., for very large values of m.

6. CONCLUSION

The computation of assignment costs is a significant contributor to the overall running time of finding a solution to the assignment problem. However, our techniques show that by utilizing lower-bound costs and pruning rules, it is possible to terminate sooner while computing fewer expensive exact costs. Our experiments show this is particularly effective in the case of driver-passenger matching in ride-hailing services, and applicable to a wide range of frameworks and applications that use the Kuhn-Munkres algorithm as a component. Moreover, the paradigm we present is generalizable and can be potentially applied to other real-world problem settings for similar benefits.

7. ACKNOWLEDGMENTS

This work was primarily conducted while Tenindra Abeywickrama was with the Grab-NUS AI Lab in the Institute of Data Science at the National University of Singapore.

8. REFERENCES

[1] How does Uber match riders with drivers? `https://www.uber.com/us/en/marketplace/matching/`.

[2] T. Abeywickrama, M. A. Cheema, and A. Khan. K-spin: Efficiently processing spatial keyword queries on road networks. *IEEE Trans. Knowl. Data Eng.*, 32(5):983–997, 2019.

[3] T. Abeywickrama, M. A. Cheema, and S. Storandt. Hierarchical graph traversal for aggregate k nearest neighbors search in road networks. In *ICAPS*, pages 2–10, 2020.

[4] T. Abeywickrama, V. Liang, and K.-L. Tan. Optimizing bipartite matching in real-world applications by incremental cost computation. *PVLDB*, 14(7):1150–1158.

[5] P. K. Agarwal and R. Sharathkumar. Approximation algorithms for bipartite matching with metric and geometric costs. In *STOC*, pages 555–564, 2014.

[6] J. Edmonds and R. M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *J. ACM*, 19(2):248–264, 1972.

[7] L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956.

[8] G. Gao, M. Xiao, and Z. Zhao. Optimal multi-taxi dispatch for mobile taxi-hailing systems. In *2016 45th International Conference on Parallel Processing (ICPP)*, pages 294–303, 2016.

[9] R. Geisberger, P. Sanders, D. Schultes, and D. Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *WEA*, pages 319–333, 2008.

[10] A. V. Goldberg and C. Harrelson. Computing the shortest path: A* search meets graph theory. In *SODA*, pages 156–165, 2005.

[11] Y. Guo, Y. Zhang, J. Yu, and X. Shen. A spatiotemporal thermo guidance based real-time online ride-hailing dispatch framework. *IEEE Access*, 8:115063–115077, 2020.

[12] R. Jonker and T. Volgenant. Improving the hungarian assignment algorithm. *Oper. Res. Lett.*, 5(4):171–175, 1986.

[13] H. W. Kuhn. The hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2(1-2):83–97, 1955.

[14] H. U. Leong, K. Mouratidis, M. L. Yiu, and N. Mamoulis. Optimal matching between spatial datasets under capacity constraints. *ACM TODS*, 35(2), 2010.

[15] J. Munkres. Algorithms for the assignment and transportation problems. *Journal of the Society for Industrial and Applied Mathematics*, 5(1):32–38, 1957.

[16] L. Ramshaw and R. E. Tarjan. A weight-scaling algorithm for min-cost imperfect matchings in bipartite graphs. In *FOCS*, pages 581–590, 2012.

[17] T. Roughgarden. Cs261: A second course in algorithms, lecture #5: Minimum-cost bipartite matching, January 2016.

[18] Y. Tang, L. H. U, Y. Cai, N. Mamoulis, and R. Cheng. Earth mover's distance based similarity search at scale. *PVLDB*, 7(4):313–324, 2013.

[19] N. Tomizawa. On some techniques useful for solution of transportation network problems. *Networks*, 1(2):173–194, 1971.

[20] I. H. Toroslu and G. íçoluk. Incremental assignment problem. *Inf. Sci.*, 177(6):1523–1529, 2007.

[21] Z. Xu, Z. Li, Q. Guan, D. Zhang, Q. Li, J. Nan, C. Liu, W. Bian, and J. Ye. Large-scale order dispatch in on-demand ride-hailing platforms: A learning and planning approach. In *SIGKDD*, pages 905–913, 2018.

[22] L. Zhang, T. Hu, Y. Min, G. Wu, J. Zhang, P. Feng, P. Gong, and J. Ye. A taxi order dispatch model based on combinatorial optimization. In *SIGKDD*, pages 2151–2159, 2017.

[23] L. Zheng, L. Chen, and J. Ye. Order dispatch in price-aware ridesharing. *PVLDB*, 11(8):853–865, 2018.

[24] R. Zhong, G. Li, K.-L. Tan, L. Zhou, and Z. Gong. G-tree: An efficient and scalable index for spatial search on road networks. *IEEE Trans. Knowl. Data Eng.*, 27(8):2175–2189, 2015.

Technical Perspective: Imperative or Functional Control Flow Handling: Why not the Best of Both Worlds?

Bill Howe
University of Washington
billhowe@uw.edu

There is a tension between an imperative style for control flow that has been shown to be easier to use, especially for novices, and a functional style for control flow that better exposes optimization opportunities, thereby making the optimizers more capable. The authors of "Efficient Control Flow in Dataflow Systems: When Ease-of-Use Meets High Performance" propose Mitos, a program rewriting framework that achieves the best of both worlds by borrowing program analysis concepts from compilers and lifting them to the distributed dataflow regime. Dataflow systems require significant data movement during processing, which can be highly redundant and wasteful in the context of iteration: naive execution plans can reprocess the same massive dataset on each iteration, and iteration $i+1$ must wait until iteration i is finished. The authors design a mechanism for labeling each intermediate result with its execution path, allowing the system to simultaneously manage complex branching situations while also implementing efficient processing via loop pipelining, all by reasoning about and comparing execution paths.

Mitos uses an intermediate representation that adapts static single assignment form (SSA) to the distributed computing reg-ime: Each variable is assigned only once; subsequent assignments to a variable are replaced with a new version. A single dataflow graph is produced by first applying simple syntactic preprocessing: unchaining sequences of calls into separate lines, then wrapping scalar assignments as bags to unify scalar and collection processing. Then, the SSA transformation divides a program into basic blocks. This process is straightforward, except that a variable assigned in two different branches (e.g., in the if-block and the else-block) will have different versions, only one of which is correct. In a compiler, the SSA transformation uses move instructions to implement ϕ-functions that make this decision — functions that determine which version of a variable to use in the presence of branching.

Mitos implements ϕ-functions and addresses a number of other control flow tasks through an elegant design: each set of intermediate results (bag) is labeled with the execution path that produced it. An execution path is a sequence of basic block labels. By reasoning about these paths, Mitos can allow a local physical operator on a single machine to process multiple branches simultaneously, as directed by the control flow messages it receives. For example, Mitos can select the latest iteration to produce a value of variable x by inspecting the length of the prefixes that end in x, and Mitos can determine which bag to work on by comparing the current path with the bag identifier.

Control flow decisions are broadcast to all machines independently of dataflow. To provide fault tolerance, Mitos allows one distinguished control flow manager (the coordinator) to decide when snapshots of basic blocks should be taken, and broadcasts these decisions to all machines who record and recover snapshots asynchronously.

The performance results are striking: multiple orders of magnitude improvements can be realized over Spark (which has no way to avoid reprocessing loop-invariant data) and Flink (which has no loop pipelining), while scaling better as well.

Previous approaches to iteration and control flow for distributed systems tended to be system-specific, such as HaLoop [2], or so general as to resist straightforward implementation in common systems, such as Blazes [1]. While some systems, including Naiad [3], offer powerful and general optimizations for iteration, they require programming in a functional style that can limit uptake and be incompatible with popular systems.

The authors' research strategy is apparent: to adapt ideas from a relevant area (compilers), and ensure interoperability for your optimizations through very general intermediate representations. Mitos is designed to work with any dataflow system that meets two modest requirements: a node in the dataflow graph must be capable of arbitrary stateful computation, and engine must support arbitrary cycles. These two requirements are met by a number of dataflow engines, including Flink, Naiad, Dandelion, and TensorFlow.

1. REFERENCES

[1] P. Alvaro, N. Conway, J. M. Hellerstein, and D. Maier. Blazes: Coordination analysis for distributed programs. In *2014 IEEE 30th International Conference on Data Engineering*, pages 52–63. IEEE, 2014.

[2] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. The haloop approach to large-scale iterative data analysis. *VLDB J.*, 21(2):169–190, 2012.

[3] D. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A timely dataflow system. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*. ACM, 2013.

Imperative or Functional Control Flow Handling: Why not the Best of Both Worlds?

Gábor E. Gévay
TU Berlin
ggab90@gmail.com

Tilmann Rabl[1]
HPI, Uni Potsdam
Tilmann.Rabl@hpi.de

Sebastian Breß[1]
Snowflake Inc.
sebastian.bress@dfki.de

Loránd Madai-Tahy[1]
mindsquare AG
Lorand.madai@gmail.com

Jorge-Arnulfo Quiané-Ruiz
TU Berlin, DFKI GmbH
jorge.quiane@tu-berlin.de

Volker Markl
TU Berlin, DFKI GmbH
Volker.Markl@dfki.de

ABSTRACT

Modern data analysis tasks often involve control flow statements, such as the iterations in PageRank and K-means. To achieve scalability, developers usually implement these tasks in distributed dataflow systems, such as Spark and Flink. Designers of such systems have to choose between providing imperative or functional control flow constructs to users. Imperative constructs are easier to use, but functional constructs are easier to compile to an efficient dataflow job. We propose Mitos, a system where control flow is both easy to use and efficient. Mitos relies on an intermediate representation based on the static single assignment form. This allows us to abstract away from specific control flow constructs and treat any imperative control flow uniformly both when building the dataflow job and when coordinating the distributed execution.

1 Introduction

Modern data analytics typically achieve scalability by relying on dataflow systems, such as Spark [23] and Flink [8]. Besides this scalability need, many data analysis algorithms require support for control flow statements. For example, many graph analysis tasks are iterative, such as PageRank and computing connected components by label propagation. Other data science pipelines are also mainly composed of iterative programs. K-means clustering and gradient descent are two commonly occurring iterative tasks. Additionally, control flow can get more complex: An iterative machine learning training task can be inside another loop for hyperparameter optimization. Nested loops also appear inside a single algorithm, such as the coloring algorithm for computing strongly connected components [18]. Programs may contain if statements inside loops, such as in simulated annealing.

However, despite that control flow statements are at the core of modern data analytics, supporting control flow is still a weak spot of dataflow systems: They either suffer from poor performance or are hard to use. On the one hand, in some systems, such as Spark, users express iterations inside the driver program, using the standard, *imperative* control flow constructs. Although this imperative approach is easy to use, it launches a new dataflow job for every iteration step, which hurts performance because of a high inherent job-launch overhead and lost optimization opportunities. On the other hand, some other systems, such as Flink and Naiad [17], provide *native control flow* support [10], i.e., users can include iterations in their (cyclic) dataflow jobs. This removes the job-launch overhead, which is present in Spark, resulting in much better performance. However, this high performance comes at a price: Users have to express iterations by calling higher-order functions, which are harder to use than the imperative control flow of Spark.

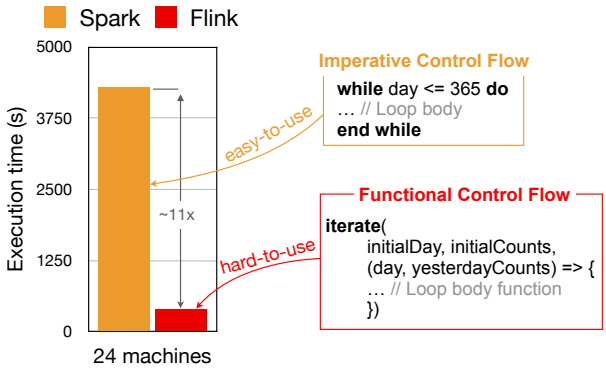


Figure 1: Imperative vs. functional control flow.

To illustrate this problem, we ran an experiment with Spark and Flink, using a program that computes the visit counts from a year of page visit logs. This program has a loop that reads a different file at each iteration step and compares the visit counts with the previous day. Figure 1 shows the results of this experiment. We observe that Spark is more than an order of magnitude slower than Flink because it does not support native iterations. Spark launches a new dataflow job for every iteration step, incurring a high overhead. However, Flink is harder to use than Spark. In Flink, users call the *iterate* higher-order function and give the loop body as an argument. The loop body is a function that builds the dataflow job fragment representing the actual loop body operations. This API is hard for non-expert

[1] Work done while the author was at TU Berlin.

 DOI: https://doi.org/10.1109/ICDE51399.2021.00127

users, such as data scientists,[2] who prefer the imperative control flow of Spark, similar to, e.g., Python, R, or Matlab.

Ideally, the system should allow users to express control flow using simple imperative control flow statements, while matching the performance of native control flow. In other words, we want a system that marries the ease-of-use of Spark with the high efficiency of Flink. This is challenging because normally a dataflow job is built from just the method calls (e.g., *map*, *join*) that the user program makes to the system. However, to build a complete cyclic dataflow job from imperative control flow, the system also needs to inspect other parts of the user code, such as the control flow statements: It also has to insert special nodes and edges into the dataflow job for such parts of the code.

We propose Mitos, a system where control flow support matches Spark's ease-of-use, and that significantly outperforms both Spark and Flink. Specifically, it outperforms Spark because of native iterations, and it outperforms Flink's native iterations because of loop pipelining. Mitos uses compile-time metaprogramming to parse an imperative user program into an intermediate representation (IR) that is based on static single assignment form (SSA). The IR abstracts away specific control flow constructs and thus facilitates the building of a single (cyclic) dataflow job from any program with imperative control flow. At runtime, Mitos coordinates the distributed execution of control flow using a novel coordination algorithm that also leverages our IR to handle any general imperative control flow. In summary, we make three major contributions:

(1) We propose a compilation approach based on metaprogramming to build a single dataflow job of a distributed dataflow system from a program with general imperative control flow statements. Specifically, we use Scala macros [7] to rewrite the user program's abstract syntax tree. (Sec. 4)

(2) We devise a mechanism that coordinates and communicates the control flow decisions between machines. The mechanism supports any imperative control flow uniformly (since it relies on the SSA representation of control flow), and enables two optimizations: loop pipelining, i.e., overlapping iteration steps, and loop-invariant hoisting, i.e., reusing loop-invariant (static) datasets during subsequent iteration steps. (Sec. 5)

(3) We experimentally evaluate Mitos using a real task and microbenchmarks. We mainly compare its performance to Flink (as a system supporting native control flow), and Spark (as a system providing ease-of-use). Mitos is more than one order of magnitude faster than Spark, and, surprisingly, it is also up to 10.5× faster than Flink. (Sec. 6)

2 Motivating Example

We now show an example to illustrate the problems of current dataflow systems with imperative control flow. Consider a program that computes the visit counts for each page per day in a year of page visit logs. Assume that the log of each day is read from a separate file, and each log entry is a page ID, which means that someone has visited the page.

[2]A simple search on stackoverflow.com for the terms *Flink iterate* or *TensorFlow while_loop* shows that many users are indeed confused by such a functional control flow API.

```
1: for day = 1 .. 365 do
2:   visits = readFile("PageVisitLog_" + day) // page IDs
3:   counts = visits.map(x => (x,1)).reduceByKey(_ + _)
4:   counts.writeFile("Counts_" + day)
5: end for
```

We cannot express this simple program in Flink's native iterations, because Flink does not support reading and writing files inside native iterations. However, not using native iterations would cause each step to launch a new dataflow job, which has an inherent high overhead (see Spark in Figure 1).

This simple program can easily become more complicated. Imagine that instead of just writing out the visit counts for each day separately, we want to compare the visit counts of consecutive days. We replace Line 4 with the following:

```
4: if day != 1 then
5:   diffs =
6:     (counts join yesterdayCounts)
7:     .map((id,today,yesterday) => abs(today - yesterday))
8:   diffs.sum.writeFile("diff" + day)
9: end if
10: yesterdayCounts = counts
```

If it is not the first day, we join the current counts with the previous day's counts (Line 6). We then compute pairwise differences (Line 7), sum up the differences (Line 8), and write the sum to a file. At the end, we save the current counts so that we can use them the next day (Line 10). We can see that it is natural to use an if statement inside the loop. On top of that, we could replace the computation of visit counts (Line 3) with a more complex computation that itself involves a loop, such as PageRank. This would result in having nested loops. Unfortunately, Flink does not provide native support for either nested loops or if statements inside loops. On the other side, Spark does not have native support for any control flow at all.

Yet, this program can become even more complex. Imagine we are interested only in a certain page type. As the logs do not contain this information, we have to read a dataset containing the page types before the loop. Inside the loop, we then add the line below before Line 3, which joins the visits and page types, and filters based on page type:

```
3: visits = (visits join pageTypes).filter(p => p.type=...)
```

It is worth noting that the *pageTypes* dataset does not change between iteration steps, i.e., it is *loop-invariant*. This clearly opens an opportunity for optimization: Even though the join method is called inside the loop, we can build the hash table of the join only once before the loop and probe it at every iteration step. This is only possible if the system implements the loop as a native iteration. This is because all iteration steps are in a single dataflow job, which enables the join operator to keep the hash table throughout the entire loop. Nevertheless, we cannot express this program using Flink's native iterations because of the aforementioned issues.

Note that iterations are at the core of machine learning training algorithms and hyperparameter search. This makes Mitos an important piece in modern analytics, such as the ones targeted by Agora [21].

3 Mitos Overview

Mitos compiles a data analysis program with imperative control flow statements into a *single* dataflow job for distributed execution on a dataflow system.

Figure 2 illustrates the general architecture of Mitos. A user provides a data analysis program in a high-level lan-

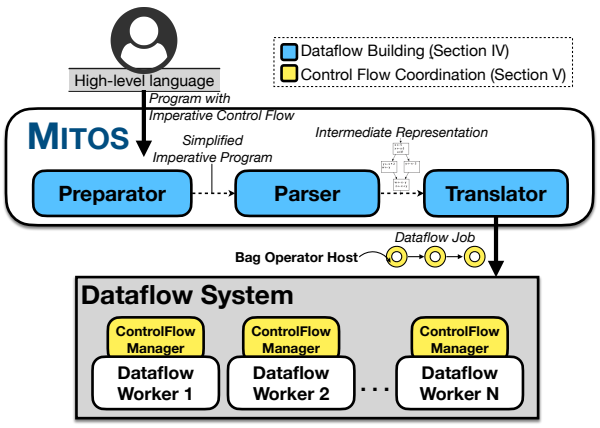


Figure 2: Mitos' architecture.

guage with imperative control flow support. The language has a scalable collection type (akin to RDD), which we call *bag* henceforth. Given an imperative program, Mitos first simplifies it to make each assignment statement have only a single bag operation (e.g., a *map*). It then parses this simplified imperative program to an intermediate representation (IR). From there, it creates a dataflow job of a distributed dataflow system (Sec. 4). Finally, Mitos sends the job for execution on the underlying dataflow system, and coordinates the distributed execution of control flow (Sec. 5).

Generality for Backends. Although we use Flink as our target dataflow system, Mitos is general: It only requires a dataflow system that allows for arbitrary stateful computations in the dataflow vertices, and supports arbitrary cycles in the dataflow graph. Examples of systems that support cycles are Flink, Naiad [17], Dandelion [20], and TensorFlow. Note that, for Mitos' loop pipelining to have a significant effect, the system should support pipelined data transfers.

Generality for Languages. Although we use the Emma language [2, 1] for Mitos, one could use other high-level data analytics languages that have imperative control flow support. Importantly, the language should provide the system with means to get information about the imperative control flow statements. In the case of Emma, this is achieved by compile-time metaprogramming. Specifically, we use Scala macros [7]. Julia [4] and Python also have the required metaprogramming capabilities. Alternatively, SystemDS [5] could also be integrated with Mitos. SystemDS' language is an *external* [1] domain-specific language, and thereby SystemML's compiler can naturally inspect the control flow.

Background (Compiler Concepts). We rely on a couple of basic compiler concepts: *static single assignment form* (SSA) and *basic blocks*. SSA [19] is often used in compilers to represent imperative control flow. When a program is in SSA, each variable has exactly one assignment statement. Another important characteristic of SSA is that it abstracts away from specific control flow constructs: The program is divided into so-called basic blocks. A basic block is a contiguous sequence of instructions with no control flow instructions, except at the end, where they conditionally jump to the beginning of a basic block. For example, consider a loop body consisting of a single basic block. The last instruction jumps either back to the beginning of the loop body block or to the basic block that is after the loop.

4 Dataflows Jobs from Imperative Programs

Our goal is to produce a *single* dataflow job from a user's imperative program that has arbitrary imperative control flow constructs. Doing so is far from being a trivial task. We need to inspect control flow statements and add extra edges. For example, in iterative algorithms, there is typically a dataflow node near the end of the loop body whose output has to be fed into the next iteration step. A more specific example is passing the current PageRanks from one step to the next. Additionally, we need to include non-bag variables into our dataflow jobs.

We leverage compile-time metaprogramming to overcome the above-mentioned challenges and hence create a dataflow job containing all the operations of an imperative program. Specifically, we leverage Scala macros [7] to inspect and rewrite the user program's abstract syntax tree. In more detail, we first simplify the imperative program (Sec. 4.1), and then parse it into an intermediate representation (Sec. 4.2). Both of these facilitate the translation of the user's program into a single dataflow job (Sec. 4.3).

4.1 Simplifying an Imperative Program

First, we split those assignment statements that have more than one operation on their right-hand side. For example, we split $b = a.map(...).filter(...)$ into two assignments: $tmp = a.map(...)$; $b = tmp.filter(...)$. For instance, Lines 8 & 9 in Figure 3a are the splitted version of Line 3 in Sec. 2.

Next, we take care of non-bag variables, e.g., an *Integer* loop counter or a *Double* learning rate. We wrap all these variables into one-element bags. This normalization step simplifies later dataflow-building by ensuring that we need to deal with only bag operations instead of introducing special cases for non-bag variables.

4.2 Intermediate Representation for General Control Flow

To handle all imperative control flow statements uniformly, Mitos transforms the program into an SSA-based IR [19]. SSA introduces a different variable for each assignment statement: if a variable in the original program had more than one assignment statement, we rename the left-hand sides of all these assignments to unique names. At the same time, we update all references to these variables with the new names. However, this updating step is not directly possible if there are different control flow paths that assign different values to a variable. In this case, the different assignments in the different control flow paths are renamed to different names and hence there is no single name to change a reference into:

```
1: if ... then
2:    a = ...
3: else
4:    a = ...
5: end if
6: b = a.map(...)
```

Note that after we change the left-hand sides of the assignments in Line 2 and 4 to different names, we cannot simply change the variable reference in Line 6 to just one of them at compile time. Therefore, we have to choose the value to refer to at runtime, based on the actual control flow path that the program execution takes. SSA makes this runtime choice explicit by introducing Φ-functions (Line 6):

```
1: if ... then
2:   a1 = ...
3: else
4:   a2 = ...
5: end if
6: a3 = Φ(a1,a2)
7: b = a3.map(...)
```

We explain how Mitos tracks the control flow and thus how Φ-functions choose between their inputs in Sec. 5.

By relying on SSA, we abstract away from specific control flow constructs, and thus handle all control flow uniformly: Control flow constructs are translated into basic blocks and conditional jumps at the end of basic blocks.

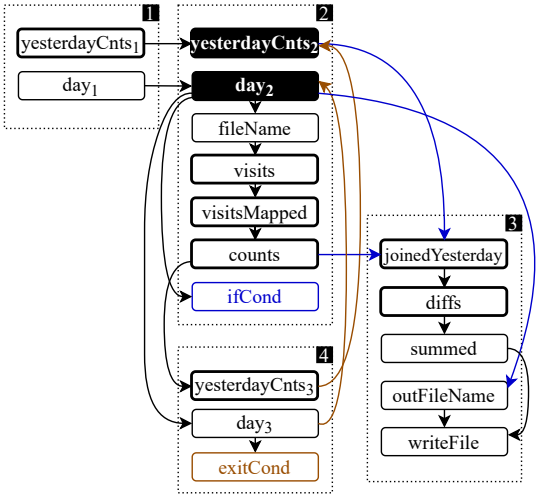
4.3 Translating an Imperative Program to a Single Dataflow

After simplifying an imperative program and putting it into our intermediate representation, the final step to build a dataflow job is now simple: We create a single dataflow node from each assignment statement and a single dataflow edge from each variable reference. For example, from $c = a$ *join* b, we create a *join* node, whose two input edges come from the nodes of the a and b variables.

To better illustrate this final translation step, we use our Visit Count running example program (Sec. 2). Figure 3a shows the program's intermediate representation, with the basic blocks as dotted rectangles, and Figure 3b shows the corresponding Mitos dataflow. Note that the join with the page types is not included for simplicity. As explained in Sec. 4.1, we wrap non-bag variables in one-element bags. We show the extra code for this in *italic* in Figure 3a. The corresponding nodes in Figure 3b have thin borders. We also create the nodes with the black background from assignments whose right-hand sides are Φ-functions (Lines 4–5). Unlike other nodes, the origins of their inputs depend on the execution path that the program has taken so far: In the first iteration step, they get their values from outside the loop (Lines 1 & 2), but then from the previous iteration step (Lines 18 & 19). This choice is represented by Φ-functions of the SSA form. The blue node corresponds to the *ifCond* variable (Line 10), and the brown node to the loop exit condition (Line 20). These *condition nodes* determine the control flow path. Edges with corresponding colors are *conditional edges*. A condition node determines whether a conditional edge with the same color transmits data in a certain iteration step, as we explain in the following section.

```
1: yesterdayCnts1 = EmptyBag                                   1
2: day1 = newBag(1)
3: do
4:   yesterdayCnts2 = Φ(yesterdayCnts1,yesterdayCnts3)        2
5:   day2 = Φ(day1,day3)
6:   fileName = day2.map(x => "pageVisitLog" + x)
7:   visits = readFile(fileName)
8:   visitsMapped = visits.map(x => (x,1))
9:   counts = visitsMapped.reduceByKey(_ + _)
10:  ifCond = day2.map(x => x != 1)
11:  if ifCond then
12:    joinedYesterday = counts join yesterdayCnts2           3
13:    diffs = joinedYesterday.map(...)
14:    summed = diffs.reduce(_ + _)
15:    outFileName = day2.map(x => "diff" + x)
16:    summed.writeFile(outFileName)
17:  end if
18:  yesterdayCnts3 = counts                                   4
19:  day3 = day2.map(x => x + 1)
20:  exitCond = day3.map(x => x ≤ 365)
21: while exitCond
```

(a)



(b)

Figure 3: (a) SSA representation of Visit Count and (b) its Mitos dataflow: The basic blocks are marked with dotted rectangles; The small rectangles are dataflow nodes, corresponding to variables in SSA; The variables corresponding to the thick-bordered nodes are bags; The colored nodes make control flow decisions and influence the same-colored edges.

5 Control Flow Coordination

Once a job is submitted for execution in an underlying dataflow system, Mitos has to coordinate the distributed execution of control flow constructs. It communicates control flow decisions between worker machines, gives appropriate input bags to operators for processing, and handles conditional edges. We achieve this via two components: the *control flow manager* and the *bag operator host*. The control flow manager communicates control flow decisions among machines (with one instance per machine). Next, each operator is wrapped inside a bag operator host, which implements the coordination logic from the operators' side. We refer to these two components together as the *Mitos runtime* (runtime, for short), and we detail them in the following.

Before diving into the runtime, we first give some preliminaries. We will use the terms "logical" and "physical" to refer to parallelization: A dataflow system parallelizes a dataflow graph (job) by creating multiple *physical* instances of each *logical* operator. A *logical edge* between two logical operators is also multiplied into *physical edges*. Note that if an operator requires a shuffle (e.g., joins), then one physical instance of the operator has many physical input edges corresponding to one logical input edge.

5.1 Challenges for the Runtime

Challenge 1. Input elements from different bags can get mixed. Mitos performs loop pipelining, i.e., different iteration steps can potentially overlap. That is, at a certain time, different operators or different physical instances of the same operator may be processing different bags that belong to different iteration steps. An example is the Visit Count program's

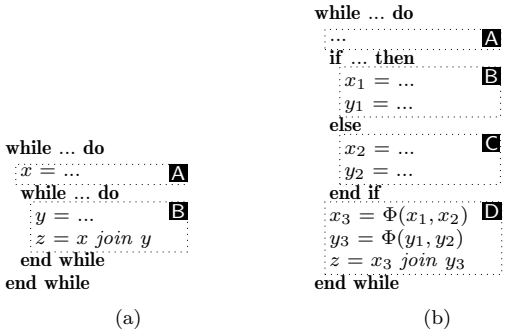


Figure 4: Programs with non-trivial control flow structures.

file reading: When any instance of the file-reading operator is done reading the file of the current iteration step, the instance can start working on the file that belongs to the next step. The difficulty is that the output from these different instances get mixed when the next operator is connected by a shuffle. This is because in case of a shuffle, each instance of the next operator receives input from all instances of the previous operator. This means that the runtime has to separate input elements that belong to different steps, so that appropriate inputs are used for computing an output bag.

Challenge 2. The matching of input bags of binary operators is not always one-to-one. In the case of binary operators (e.g., join), the runtime gives a pair of bags to an operator at a time. To form a pair, we have to match bags arriving on one logical input edge to bags arriving on the other logical input edge. This matching is not always one-to-one, e.g., sometimes one bag has to be used several times, each time matching it with a different bag. The example program in Figure 4a demonstrates such a case. Input x of the join is from outside the loop, while input y is from inside the loop. This means that when the runtime provides the join with pairs of input bags, it has to use a bag from x several times, matching it with different bags from y each time.

Challenge 3. First-come-first-served does not work for choosing the input bags to process. Even when the matching of bags between the two logical input edges is one-to-one, the following naive algorithm for matching them up does not work: Assume we order bags in the same order as their first elements arrive. In this case, we could match bags from each of the inputs in the order they arrived, i.e., match the first bag from one input with the first bag from the other input, then match the second bags from both inputs, and so on. However, doing so might lead to errors. Suppose that the control flow in Figure 4b reaches the basic blocks in the following order: $ABDACD$. It is then possible that, due to irregular processing delays, the operator of x_3 gets data from x_1 first and then from x_2, while the operator of y_3 gets data from y_2 first and then from y_1. This can happen because the operators in the different *if* branches are not synchronized, i.e., they do not agree on a global order in which to process bags. This would clearly lead to an incorrect result: The operator of z has to match the bag that originates from x_1 with the bag that originates from y_1, and match the bag that originates from x_2 with the bag that originates from y_2. Note that this issue can arise only if we perform loop pipelining. Otherwise, all operators finish the processing of one step before any operator starts the next step.

5.2 Coordination Based on Bag Identifiers

We tackle the aforementioned challenges by introducing a *bag identifier* (Sec. 5.2.1). We make sure that the same bags and same bag identifiers are created during the distributed execution as they would be in a non-parallel execution. More specifically, we show how a physical operator instance can determine during a distributed execution: (i) the identifier of the output bag that it should compute next (Sec. 5.2.2); (ii) the identifier of the input bags that it should use to compute a particular output bag (Sec. 5.2.3), and; (iii) on which conditional output edge it should send a particular output bag (Sec. 5.2.4). Note that the Mitos runtime is designed for allowing operators to start computing an output bag as soon as its inputs start to arrive. This enables loop pipelining, i.e., an operator can start a later step while other operators are still working on a previous step.

5.2.1 Bag Identifiers with Execution Paths

A bag identifier encapsulates both the identifier of the logical operator that created the bag and the execution path of the program up to the creation of the bag. The execution path is a sequence of basic blocks that the execution reached. In a distributed execution, the execution path is determined by the condition nodes. A condition node appends a basic block to the path when it evaluates its condition. Condition nodes let all other operators know about these decisions through the control flow manager. The local control flow manager broadcasts the decision to all remote control flow managers through TCP connections (which are independent from dataflow edges). This way every physical instance of every operator knows how the execution path evolves. The bag identifiers are also used to separate elements that belong to different bags (Challenge 1): we tag each element with the bag identifier that it belongs to.

5.2.2 Choosing Output Bags

By watching how the execution path evolves, operators can choose the identifiers of output bags to be computed: When the path reaches the basic block of the operator, the operator starts to compute the bag whose bag identifier contains the current path. For example, in Challenge 3, this means that the physical operator instances of both x_3 and y_3 choose to compute the output bag with path ABD in its identifier first, and then $ABDACD$.

5.2.3 Choosing Input Bags

When an operator O_2 decides to produce a particular output bag g_2 next, it also needs to choose input bags for it (Challenges 2 & 3). This choice is made independently for each logical input.

In a non-parallel execution, the operator would use the latest bag that was written to the variable that the particular input refers to. We mirror this behavior in the distributed execution, by examining the execution path while keeping in mind the operator's and input's basic blocks. More specifically, for a logical input i of O_2, let O_1 be the operator whose output is connected to i, b_1 and b_2 be the basic blocks of O_1 and O_2, and c be the execution path in the identifier of g_2. To determine the identifier of a bag coming from i to compute an output bag g_2, we consider all the prefixes of c. Among these prefixes, we choose the longest one such that it ends with b_1. For example, in Figure 4a when we are computing z and choosing an input bag from x, we always choose the bag that the latest run of the outer loop com-

puted. Concretely, if we are computing the bag with the path $ABBABBB$, then the prefix we choose is $ABBA$.

5.2.4 Choosing Conditional Outputs

Operators look at how the execution path evolves after a particular output bag, and send the bag on such conditional output edges whose target is reached by the path before the next output bag is computed. Specifically, let O_1 be an operator that is computing output bag g, e be a conditional output edge of O_1, O_2 be the operator that is the target of e, b_1 be the basic block of O_1, b_2 be the basic block of O_2, and c be the execution path of the identifier of g. Note that the last element of c is b_1. O_1 should examine each new basic block appended to the execution path and send g to O_2 when the path reaches b_2 for the first time after c but before it reaches b_1 again. This means that instances of O_1 can discard their partitions of g once the execution path reaches such a basic block from which every path to b_2 on the control flow graph goes through b_1.

5.3 Optimization: Loop-Invariant Hoisting

We now show how to incorporate loop-invariant hoisting into our dataflows. That is, we show how to improve performance when an iteration involves a loop-invariant (static) dataset, which is reused without updates during subsequent iteration steps. We can see an example of this in our running example in Sec. 2: The *pageTypes* dataset is read from a file outside the iteration and is used in a join inside the iteration. Another example is any iterative graph algorithm that joins with a static dataset containing the graph edges.

It is a common optimization to pull those parts of a loop body that depend on only static datasets outside of the loop, and thus execute them only once [10, 6, 9]. However, launching new dataflow jobs for every iteration step prevents this optimization in the case of binary operators where only one input is static. For example, if a static dataset is used as the build-side of a hash join, then the system should not rebuild the hash table at every iteration step. Mitos operators can keep such a hash table in their internal states across iteration steps. We make this possible by having a single dataflow job, where operator lifetimes span all the steps.

We now show how to incorporate this optimization into Mitos. Normally, the bag operators drop the state that they have built up during the computation of a specific output bag. However, to perform loop-invariant hoisting, the runtime lets the bag operators know when to keep their state that they build up for an input (e.g., the hash table of a hash join). Assume, without loss of generality, that the first input of the bag operator is the one that does not always change between output bags, and the second input changes for every output bag. Between two output bags, the runtime tells the operator whether the next bag coming from the first input changes for the next output bag. If it changes, the operator should drop the state built up for the first input. Otherwise, the operator implementation should assume that the first input is the same bag as before. For our example in Figure 4a, the first input bag changes at every step of the outer loop, but not between steps of the inner loop.

6 Evaluation

We implemented Mitos on Java 8 and Scala 2.11 and used Flink 1.6 as an underlying dataflow system.

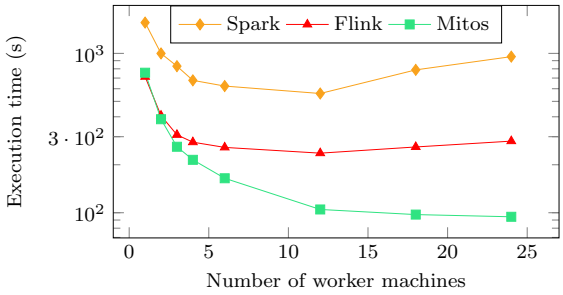


Figure 5: Strong scaling for Visit Count.

6.1 Setup

Hardware. We ran our experiments on a cluster of 26 machines, each with 2×8-core AMD Opteron 6128 CPUs, 32 GB memory, 4×1 TB disks, Gigabit Ethernet, on Ubuntu 18.04.
Tasks and Datasets. We used the Visit Count example introduced in Sec. 2, where we compare visit counts of subsequent days. We used two versions: one with and one without the join of the *pageTypes* dataset. We have generated random inputs, with the visits uniformly distributed.
Baselines. We performed most of our experiments against Spark 3.0 and Flink 1.6, with both running on OpenJDK 8. We stored input data on HDFS 2.7.1. We also performed microbenchmarks against Naiad [17] and TensorFlow [22].
Repeatability. We report numbers for the average of three runs. We also provide the code for Mitos[3].

6.2 Strong Scaling

We start by evaluating how well Mitos scales with respect to the number of worker machines, and how it performs vis-a-vis two state-of-the-art systems: Spark and Flink.

Figure 5 shows the results for the Visit Count task. The size of the input for one day is 21 MB, and there are 365 days, i.e., the total input size is 7.6 GB. We observe that Mitos scales gracefully with the number of machines. However, Spark and Flink show a surprising *increase* in execution time as we give more machines to the system. This is because of their overhead in each iteration step increases with the number of machines, and thereby becoming a dominant factor in the execution time. We study this iteration overhead in Sec. 6.4. In particular, we observe that with the maximum number of machines, Mitos is 10× faster than Spark and 3× faster than Flink. The latter is an interesting result as Flink provides native control flow support. Our system improves over Flink because it performs loop pipelining.

6.3 Scalability With Respect to Input Size

Our goal is now to analyze how well Mitos performs with different input dataset sizes for Visit Count. Figure 6 shows the results of this experiment. We observe that our system significantly outperforms Spark and the performance gap increases with the dataset size: it goes from 23× to more than two orders of magnitude. This is because of the loop-invariant hoisting optimization (see Sec. 6.5 for a detailed evaluation). Mitos outperforms also Flink, by 3.1–10.5×, while being easier to use due to its imperative control flow interface. The surprisingly large improvement factor over Flink for small data sizes is due to Flink's native iteration having a large per-step overhead due to a technical issue[4].

[3] `https://github.com/ggevay/mitos`
[4] `https://issues.apache.org/jira/browse/FLINK-3322`

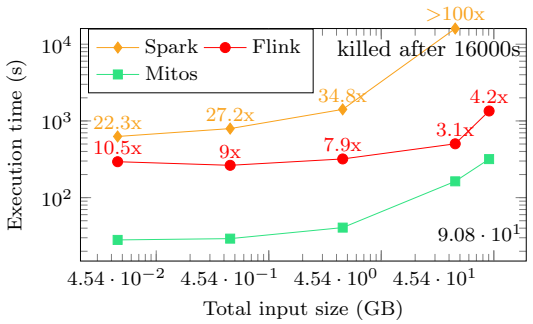


Figure 6: Visit Count (with the *pageTypes* dataset) when varying the input size. The factors are relative to Mitos.

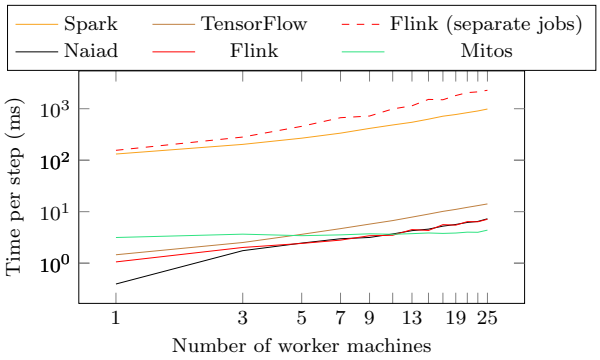


Figure 7: Log-log plot for the per-step overhead.

6.4 Iteration Step Overhead

We isolate the step overhead from the actual data processing in a microbenchmark: a simple loop with minimal actual data processing in each step. In this experiment, we also considered TensorFlow and Naiad as baselines to better evaluate the efficiency of Mitos. Figure 7 shows the results. We observe that the native iteration of Mitos is about two orders of magnitude faster than launching new jobs for each step, i.e., Spark and Flink (separated jobs). It is interesting to note that the job launch overhead increases linearly with the number of machines. Importantly, this means that scaling out to more machines makes the step overhead problem of Spark worse. Furthermore, we can also see that Mitos matches the performance of other systems with native iterations, i.e., Flink, TensorFlow, and Naiad, despite being able to handle more general control flow.

6.5 Loop-Invariant Hoisting

We proceed to evaluate the loop-invariant hoisting optimization in Mitos. For this, we used the version of the Visit Count example that has the join with the loop-invariant *pageTypes* dataset at every iteration step. Figure 8 shows the results when varying the size of the loop-invariant dataset, while keeping the other part of the input constant (13 GB). We observe that increasing the loop-invariant dataset size has very little effect on Mitos and Flink. This is because they perform the loop-invariant hoisting optimizations i.e., they build the hash table for the join only once and then just probe the hash table at every iteration step.

On the other hand, the execution time of Spark linearly increases because Spark does not perform this loop-invariant hoisting optimization. As a result, Mitos is up to 45× faster

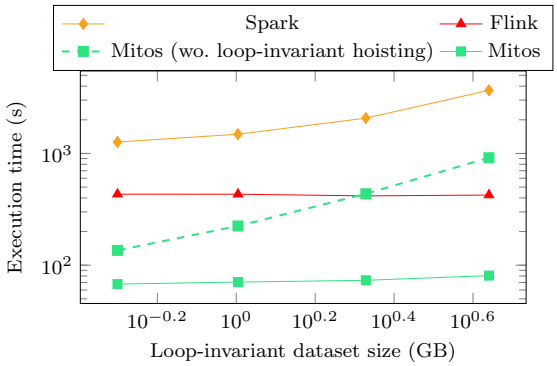


Figure 8: Varying the loop-invariant dataset size.

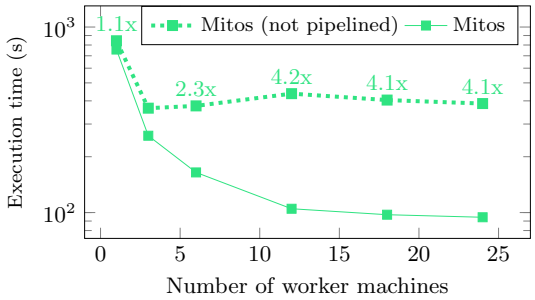


Figure 9: Loop pipelining with varying machine count.

than Spark. Note that, in our Spark implementation, we manually inserted a repartitioning of the *pageTypes* dataset once before the loop.

To isolate the effect of loop-invariant hoisting from other differences between Spark and Mitos, we also ran Mitos with loop-invariant hoisting switched off. In this case, its execution time increases linearly with the size of the loop-invariant dataset, similarly to Spark. Therefore, Mitos is up to 11× faster than Mitos without loop-invariant hoisting.

6.6 Loop Pipelining

We now analyze the loop pipelining feature of Mitos, which allows it to outperform Flink. Recall that, even though Flink also provides native iteration support, our system is up to 3× faster in Figure 5, 3.1–10.5× faster in Figure 6, and 5–6× faster in Figure 8. As one might think that this performance difference could come from other factors, we ran an experiment to better isolate the effect of loop pipelining. We ran Visit Count (without the *pageTypes* dataset) in Mitos with and without the loop pipelining optimization. Figure 9 shows the results. Overall, we clearly observe the benefits of loop pipelining: Our system can be up to 4× faster with than without loop pipelining, which is made possible by our control flow coordination mechanism.

7 Related Work

Arvind et al. [3] include control flow into dataflow graphs through the *switch* and *merge* primitives (operations), which TensorFlow recently adopted [22]. Mitos, in contrast to TensorFlow, applies to general data analytics in addition to machine learning. The recent AutoGraph [16] and Janus [15] systems compile imperative control flow to TensorFlow, which makes them not directly applicable for general data analytics. Hirn et al. [13, 14] compile from PL/SQL's imperative

control flow to recursive SQL queries. Gévay et al. [12] survey the literature on how various distributed dataflow systems handle control flow. Matryoshka [11] adds control flow support to the flattening technique for nested parallelism.

Several systems natively support a limited number of control flow constructs, such as Flink [10], and Naiad [17]. However, they rely on functional-style APIs, where each control flow construct is a higher-order function. For example, in TensorFlow, users call the *while_loop* method and provide two functions: one for building the dataflow of the loop body and another for building the dataflow of the loop exit condition. Similarly, in Flink, users call the *iterate* method and supply the loop body as a function that builds the dataflow job fragment representing the loop body. A simple search for these Flink and TensorFlow methods on `stackoverflow.com` shows many users being confused by this API. Mitos allows users to write imperative control flow statements, such as regular while-loops and if statements, which makes it more accessible.

Other works have added iteration to systems that do not support control flow natively. HaLoop [6] and Twister [9] extend MapReduce to provide support for iterations. Nonetheless, in contrast to Mitos, the programming model of these systems is directly based on MapReduce rather than building complex programs using a collection-based API.

Although loop-invariant hoisting is a well-known optimization in the context of distributed data analytics [10, 17, 6, 9], none of these works supports programs with imperative control flow constructs. SystemDS [5] does, but it cannot perform it on a binary operator having only one static input, e.g., the hash join that we used in Sec. 5.3.

8 Conclusion

Modern data analysis requires complex control flow constructs, yet dataflow systems either suffer from poor performance for programs with control flow or are hard to use. We presented Mitos, a system that allows users to express control flow by easy-to-use imperative constructs, and still executes these programs efficiently as a single dataflow job. Mitos uses an intermediate representation based on SSA, which abstracts away from specific control flow constructs. Relying on SSA allows us to handle all imperative control flow in a uniform way, both when building a dataflow job, and when coordinating the distributed execution of control flow statements. Our coordination mechanism enables loop pipelining and loop-invariant hoisting.

Acknowledgments

We thank Alexander Alexandrov for pointing our attention to SSA, and Eleni Tzirita Zacharatou for the system name. This work was funded by the German Ministry for Education and Research as BIFOLD – Berlin Institute for the Foundations of Learning and Data (ref. 01IS18025A and ref. 01IS18037A), and German Research Foundation – Project-ID 414984028 – SFB 1404.

9 References

[1] A. Alexandrov, G. Krastev, and V. Markl. Representations and optimizations for embedded parallel dataflow languages. *TODS*, 44(1):4, 2019.

[2] A. Alexandrov, A. Kunft, A. Katsifodimos, F. Schüler, L. Thamsen, O. Kao, T. Herb, and V. Markl. Implicit parallelism through deep language embedding. In *SIGMOD*. ACM, 2015.

[3] Arvind and D. E. Culler. Dataflow architectures. *Annual review of computer science*, 1(1), 1986.

[4] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah. Julia: A fresh approach to numerical computing. *SIAM review*, 59(1):65–98, 2017.

[5] M. Boehm, I. Antonov, S. Baunsgaard, M. Dokter, R. Ginthör, K. Innerebner, F. Klezin, S. Lindstaedt, A. Phani, B. Rath, B. Reinwald, S. Siddiqi, and S. B. Wrede. SystemDS: A declarative machine learning system for the end-to-end data science lifecycle. In *CIDR*, 2020.

[6] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. HaLoop: efficient iterative data processing on large clusters. *VLDB*, 2010.

[7] E. Burmako. Scala macros: let our powers combine!: on how rich syntax and static types work with metaprogramming. In *Proceedings of the 4th Workshop on Scala*. ACM, 2013.

[8] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache Flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[9] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox. Twister: a runtime for iterative MapReduce. In *19th ACM international symposium on high performance distributed computing*. ACM, 2010.

[10] S. Ewen, K. Tzoumas, M. Kaufmann, and V. Markl. Spinning fast iterative data flows. *Proceedings of the VLDB Endowment*, 5, 2012.

[11] G. E. Gévay, J.-A. Quiané-Ruiz, and V. Markl. The power of nested parallelism in big data processing–hitting three flies with one slap–. In *SIGMOD*, pages 605–618, 2021.

[12] G. E. Gévay, J. Soto, and V. Markl. Handling iterations in distributed dataflow systems. *ACM Computing Surveys (CSUR)*, 54(9):1–38, 2021.

[13] D. Hirn and T. Grust. PL/SQL without the PL. In *SIGMOD*, pages 2677–2680, 2020.

[14] D. Hirn and T. Grust. One with recursive is worth many GOTOs. In *SIGMOD*, pages 723–735, 2021.

[15] E. Jeong, S. Cho, G.-I. Yu, J. S. Jeong, D.-J. Shin, and B.-G. Chun. JANUS: Fast and flexible deep learning via symbolic graph execution of imperative programs. In *NSDI*. USENIX Association, 2019.

[16] D. Moldovan, J. Decker, F. Wang, A. Johnson, B. Lee, Z. Nado, D. Sculley, T. Rompf, and A. B. Wiltschko. AutoGraph: Imperative-style coding with graph-based performance. In *SysML*, 2019.

[17] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: a timely dataflow system. In *SOSP*. ACM, 2013.

[18] S. M. Orzan. *On distributed verification and verified distribution*. PhD thesis, Vrije Universiteit Amsterdam, 2004.

[19] F. Rastello. *SSA-based Compiler Design*. Springer, 2016.

[20] C. J. Rossbach, Y. Yu, J. Currey, J.-P. Martin, and D. Fetterly. Dandelion: a compiler and runtime for heterogeneous systems. In *SOSP*, pages 49–68, 2013.

[21] J. Traub, Z. Kaoudi, J.-A. Quiané-Ruiz, and V. Markl. Agora: Bringing together datasets, algorithms, models and more in a unified ecosystem [vision]. *SIGMOD Record*, 49(4):6–11, 2020.

[22] Y. Yu, M. Abadi, P. Barham, E. Brevdo, M. Burrows, A. Davis, J. Dean, S. Ghemawat, T. Harley, P. Hawkins, et al. Dynamic control flow in large-scale machine learning. In *EuroSys*, page 18. ACM, 2018.

[23] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. *HotCloud*, 10, 2010.

Technical Perspective: Relative Error Streaming Quantiles

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

The paper *Relative Error Streaming Quantiles*, by Graham Cormode, Zohar Karnin, Edo Liberty, Justin Thaler and Pavel Veselý studies a fundamental question in data stream processing, namely how to maintain information about the distribution of data in the form of *quantiles*. More precisely, given a stream S of elements from some ordered universe $\mathcal{U}$ we wish to maintain a compact summary data structure that allows us to estimate the number of elements in the stream that are smaller than a given query element $y \in \mathcal{U}$, i.e., estimate the *rank* of y. Solutions to this problem have numerous applications in large-scale data analysis and can potentially be used for range query selectivity estimation in database engines.

The challenge is to make the size s of the summary data structure (also known as a *sketch*) as small as possible while bounding the error on rank queries. If data is available in sorted order and has a known size n, an optimal solution is to store s *quantiles*, that is, elements in S with specific ranks. If S contains n elements we can store the elements in S that have rank $n/(s+1)$, $2n/(s+1), \ldots, sn/(s+1)$, which would allow us to approximate the rank of any query element $y \in \mathcal{U}$ up to an additive error of $n/(s+1)$. (Here we ignore rounding issues for simplicity.) In other words, to achieve additive error εn it suffices to use space $1/\varepsilon$.

Surprisingly, it is possible to achieve almost the same space efficiency in the setting where elements are presented in arbitrary order, and no bound on n is known. This was established in a series of papers culminating in the work of Karnin, Lang, and Liberty [4]. The simplest version of their data structure, now often referred to as the "KLL sketch" has become widely used in data processing pipelines. The popularity may be partly due to another property of the KLL sketch: Sketches for datasets S_1 and S_2 can be combined to form a sketch for the multiset union of the data sets. This property, referred to as *mergeability* means that computation of KLL sketches can be efficiently carried out in parallel and distributed settings with little communication.

In *Relative Error Streaming Quantiles*, which appeared in PODS '21 [2], the authors consider the situation where more precise information about the *tails* of the distribution (i.e., the largest and smallest elements) must be maintained. This setting is important because many data distributions have few elements in the tails, and thus the KLL sketch may have little or no information about the distribution of the data in the tails. A natural way of asking for more precise information on elements on the lower tail is to ask for a rank error guarantee that is *multiplicative*, i.e., a rank estimate that is within some factor $1+\varepsilon$ from the true rank. In contrast, the KLL sketch offers an *additive* error guarantee, which is much weaker for elements with rank $\ll n$. When a multiplicative rank guarantee is possible, by symmetry, improved guarantees can also be achieved for query elements with rank close to n.

Though multiplicative error is a desirable property, past solutions were significantly less efficient than the KLL sketch. A gap in performance is unavoidable for a broad class of algorithms, as shown in a lower bound of Cormode and Veselý [3]. Still, past solutions required larger overheads in the dependence on the stream length n or the approximation parameter ε. To improve on this, the authors go back to the KLL sketch and describe a simple but clever modification that improves past results and in fact nearly matches the lower bound. Though the algorithm, called *ReqSketch*, is relatively simple, analyzing it is not and requires ideas that go significantly beyond those needed for the KLL sketch.

The new, mergeable sketch, is not only a great theoretical contribution — it is already available along with the KLL sketch as part of the Apache Datasketches library of streaming algorithms [1] and is ready to impact computing practice!

1. REFERENCES

[1] Apache datasketches. `https://datasketches.apache.org/`. Accessed: 2022-03-10.

[2] G. Cormode, Z. S. Karnin, E. Liberty, J. Thaler, and P. Veselý. Relative error streaming quantiles. In *Proceedings of the Symposium on Principles of Database Systems (PODS)*, pages 96–108. ACM, 2021.

[3] G. Cormode and P. Veselý. A tight lower bound for comparison-based quantile summaries. In *Proceedings of Symposium on Principles of Database Systems (PODS)*, pages 81–93. ACM, 2020.

[4] Z. S. Karnin, K. J. Lang, and E. Liberty. Optimal quantile approximation in streams. In *Proceedings of Symposium on Foundations of Computer Science (FOCS)*, pages 71–78. IEEE Computer Society, 2016.

Relative Error Streaming Quantiles

[Extended Abstract]

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

Zohar Karnin
Amazon, USA
zkarnin@gmail.com

Edo Liberty
Pinecone
San Mateo, CA, USA
edo@edoliberty.com

Justin Thaler
Georgetown University
Washington, D.C., USA
justin.thaler@georgetown.edu

Pavel Veselý
Charles University
Prague, Czech Republic
vesely@iuuk.mff.cuni.cz

ABSTRACT

Estimating ranks, quantiles, and distributions over streaming data is a central task in data analysis and monitoring. Given a stream of n items from a data universe equipped with a total order, the task is to compute a sketch (data structure) of size polylogarithmic in n. Given the sketch and a query item y, one should be able to approximate its rank in the stream, i.e., the number of stream elements smaller than or equal to y.

Most works to date focused on additive εn error approximation, culminating in the KLL sketch that achieved optimal asymptotic behavior. This paper investigates *multiplicative* $(1 \pm \varepsilon)$-error approximations to the rank. Practical motivation for multiplicative error stems from demands to understand the tails of distributions, and hence for sketches to be more accurate near extreme values.

The most space-efficient algorithms due to prior work store either $O(\log(\varepsilon^2 n)/\varepsilon^2)$ or $O(\log^3(\varepsilon n)/\varepsilon)$ universe items. We present a randomized sketch storing $O(\log^{1.5}(\varepsilon n)/\varepsilon)$ items, which is within an $O(\sqrt{\log(\varepsilon n)})$ factor of optimal. Our algorithm does not require prior knowledge of the stream length and is fully mergeable, rendering it suitable for parallel and distributed computing environments.

1. INTRODUCTION

Understanding the distribution of data is a fundamental task in data monitoring and analysis. In many settings, we want to understand the cumulative distribution function (CDF) of a large number of observations, for instance, to identify anomalies. In other words, we would like to track the median, percentiles, and more generally quantiles of a massive input in a small space, without storing all the observations. Although memory constraints make an exact computation of such order statistics impossible [22], most applications can be satisfied with approximating the quantiles, which also yields a compact function with a bounded distance from the true CDF.

The problem of streaming quantile approximation captures this task in the context of massive or distributed datasets. Let $\sigma = (x_1, \ldots, x_n)$ be a stream of items, all drawn from a data universe $\mathcal{U}$ equipped with a total order. For any $y \in \mathcal{U}$, let $\mathrm{R}(y;\sigma) = \left|\left\{i \in \{1,\ldots,n\} \mid x_i \le y\right\}\right|$ be the rank of y in the stream. When σ is clear from context, we write $\mathrm{R}(y)$. The objective is to process the stream in one pass while storing a small number of items, and then use those to approximate $\mathrm{R}(y)$ for any $y \in \mathcal{U}$. A guarantee for an approximation $\hat{\mathrm{R}}(y)$ is *additive* if $|\hat{\mathrm{R}}(y) - \mathrm{R}(y)| \le \varepsilon n$, and *multiplicative* or *relative* if $|\hat{\mathrm{R}}(y) - \mathrm{R}(y)| \le \varepsilon\,\mathrm{R}(y)$. Estimating ranks immediately yields approximate quantiles, and vice versa, with a similar error guarantee (recall that a ϕ-quantile for $\phi \in [0,1]$ is the $\lfloor \phi n \rfloor$'th smallest item in σ). We stress that we do not assume any particular data distribution or that the stream is randomly-ordered.

A long line of work has focused on achieving additive error guarantees [23, 2, 19, 24, 13, 3, 12, 1, 11, 16]. However, additive error is not appropriate for many applications. Indeed, often the primary purpose of computing quantiles is to understand the tails of the data distribution. When $\mathrm{R}(y) \ll n$, a multiplicative guarantee is much more accurate and thus harder to obtain. As pointed out by Cormode et al. [5], a solution to this problem would also yield high accuracy when $n - \mathrm{R}(y) \ll n$, by running the same algorithm with the reversed total ordering (simply negating the comparator).

A quintessential application that demands relative error is monitoring network latencies. In practice, one often tracks response time percentiles 50, 90, 99, 99.9, etc. This is because latencies are heavily long-tailed. For example, Masson et al. [21] report that for web response times, the 98.5th percentile can be as small as 2 seconds while the 99.5th percentile can be as large as 20 seconds. These unusually long response times affect network dynamics [5] and are problematic for users. Furthermore, as argued by Tene in his talk about measuring latency [27], one needs to look at extreme percentiles such as 99.995 to determine the latency such that only about 1% of users experience a larger latency during a web session with several page loads. Hence, highly accurate rank approximations are required for items y whose rank is

This is a minor revision of the paper entitled *Relative Error Streaming Quantiles*, published in PODS '21, ISBN 978-1-4503-8381-3/21/06, June 20–25, 2021, Virtual Event, China. DOI: https://dl.acm.org/doi/10.1145/3452021.3458323.

very large ($n - \mathrm{R}(y) \ll n$); this is precisely the requirement captured by the multiplicative error guarantee.

Achieving multiplicative guarantees is known to be *strictly* harder than additive ones. There are comparison-based additive error algorithms that store just $\Theta(\varepsilon^{-1})$ items for constant failure probability [16], which is optimal. In particular, to achieve additive error, the number of items stored may be independent of the stream length n. In contrast, any algorithm achieving multiplicative error must store $\Omega(\varepsilon^{-1} \cdot \log(\varepsilon n))$ items (see [5, Theorem 2]).

Remark 1. *Intuitively, the reason additive-error sketches can achieve space independent of the stream length is because they can take a subsample of the stream of size about $\Theta(\varepsilon^{-2})$ and then sketch the subsample. For any fixed item, the additive error to its rank introduced by sampling is at most εn with high probability. When multiplicative error is required, one cannot subsample the input: for low-ranked items, the multiplicative error introduced by sampling will, with high probability, not be bounded by any constant.*

The best known algorithms achieving multiplicative error guarantees are as follows. Zhang et al. [29] give a randomized algorithm storing $O(\varepsilon^{-2} \cdot \log(\varepsilon^2 n))$ universe items. This is essentially a ε^{-1} factor away from the aforementioned lower bound. There is also an algorithm of Cormode et al. [6] that stores $O(\varepsilon^{-1} \cdot \log(\varepsilon n) \cdot \log |\mathcal{U}|)$ items. However, this algorithm requires prior knowledge of the data universe $\mathcal{U}$ (since it builds a binary tree over $\mathcal{U}$), and is inapplicable when $\mathcal{U}$ is huge or even unbounded (e.g., if the data can take arbitrary real values). Finally, Zhang and Wang [28] designed a deterministic algorithm requiring $O(\varepsilon^{-1} \cdot \log^3(\varepsilon n))$ space. Recent work of Cormode and Veselý [8] proves an $\Omega(\varepsilon^{-1} \cdot \log^2(\varepsilon n))$ lower bound for deterministic comparison-based algorithms, which is within a $\log(\varepsilon n)$ factor of Zhang and Wang's upper bound.

Despite both the practical and theoretical importance of multiplicative error (which is arguably an even more natural goal than additive error), there has been no progress on upper bounds, i.e., no new algorithms, since 2007.

In this work, we give a randomized algorithm that maintains the optimal linear dependence on $1/\varepsilon$ achieved by Zhang and Wang, with a significantly improved dependence on the stream length. Namely, we design a one-pass streaming algorithm that given $\varepsilon > 0$, computes a sketch consisting of $O\left(\varepsilon^{-1} \cdot \log^{1.5}(\varepsilon n)\right)$ universe items, from which one can derive rank or quantile estimates satisfying the relative error guarantee with constant probability (see Theorem 1 for a more precise statement). Ours is the first algorithm to be strictly more space efficient than *any* deterministic comparison-based algorithm (owing to the $\Omega(\varepsilon^{-1} \log^2(\varepsilon n))$ lower bound in [8]) and is within an $O(\sqrt{\log(\varepsilon n)})$ factor of the known lower bound for randomized algorithms achieving multiplicative error. Furthermore, it only accesses items through comparisons, i.e., is comparison-based, rendering it suitable, e.g., for floating-point numbers or strings ordered lexicographically. Finally, our algorithm processes the input stream efficiently, namely, its amortized update time is a logarithm of the space bound, i.e., $O\left(\log(\varepsilon^{-1}) + \log\log(n)\right)$.

Mergeability. The ability to merge sketches of different streams to get an accurate sketch for the concatenation of the streams is highly significant both in theory [1] and in practice [25]. Such mergeable summaries enable trivial, automatic parallelization and distribution of processing massive data sets, by splitting the data up into pieces, summarizing each piece separately, and then merging the results in an arbitrary way. We say that a sketch is *fully mergeable* if building it using any sequence of merge operations (executed on singleton items) leads to the same guarantees as if the entire data set had been processed as a single stream.

The following theorem is the main result of this paper. We stress that our algorithm, which we call ReqSketch, does *not* require any advance knowledge about n, the total size of the input, which indeed may not be available in many applications.

Theorem 1. *Given parameters $0 < \delta \leq 0.5$ and $0 < \varepsilon \leq 1$, there is a randomized, comparison-based, one-pass streaming algorithm that, when processing a data stream consisting of n items from a totally-ordered universe $\mathcal{U}$, produces a summary S satisfying the following property. Given S, for any $y \in \mathcal{U}$ one can get an estimate $\hat{\mathrm{R}}(y)$ of $\mathrm{R}(y)$ such that*

$$\Pr\left[|\hat{\mathrm{R}}(y) - \mathrm{R}(y)| \geq \varepsilon\, \mathrm{R}(y)\right] < \delta\,,$$

where the probability is over the internal randomness of the streaming algorithm. If $\varepsilon \leq \sqrt{\ln \frac{1}{\delta} / \log_2(\varepsilon n)}$, then the size of S in memory words is

$$O\left(\varepsilon^{-1} \cdot \log^{1.5}(\varepsilon n) \cdot \sqrt{\log\left(\frac{1}{\delta}\right)}\right);$$

otherwise, storing S takes $O\left(\log^2(\varepsilon n)\right)$ memory words. Moreover, the summary produced is fully mergeable.

The space bound for the case $\varepsilon \leq \sqrt{\ln \frac{1}{\delta} / \log_2(\varepsilon n)}$ certainly applies for values of ε and n encountered in practice (e.g., for $n \leq 2^{64}$ and $\delta \leq 1/e$, this latter requirement is implied by $\varepsilon \leq 1/8$). A straightforward corollary of Theorem 1 is a space-efficient algorithm whose estimates are simultaneously accurate for *all* $y \in \mathcal{U}$ with high probability, while the space complexity increases by a small factor only. This follows from a standard use of the union bound with an epsilon-net argument (using failure probability $\delta' = \varepsilon\delta / \log(\varepsilon n)$).

There is also an alternative analysis of our algorithm (building on an idea from [16]), that shows a space bound of $O\left(\varepsilon^{-1} \cdot \log^2(\varepsilon n) \cdot \log\log(1/\delta)\right)$; note the exponentially better dependence on $1/\delta$, compared to Theorem 1, which, however, comes at the expense of the exponent of $\log(\varepsilon n)$ increasing from 1.5 to 2. This analysis also implies a deterministic space bound of $O(\varepsilon^{-1} \cdot \log^3(\varepsilon n))$, matching the state-of-the-art result in [28].

A proof-of-concept Python implementation of ReqSketch is available at GitHub [17] and a production-quality implementation is available in the DataSketches library [25].

1.1 Technical Overview

A starting point of the design of our algorithm is the KLL sketch [16] that achieves optimal accuracy-space trade-off for the additive error guarantee. The basic building block of the algorithm is a buffer, called a *compactor*, that receives an input stream of n items and outputs a stream of at most $n/2$ items, meant to "approximate" the input stream. The buffer simply stores items and once it is full, we sort the buffer, output all items stored at either odd or even indexes (with odd vs. even selected via an unbiased coin flip), and

clear the contents of the buffer—this is called the *compaction operation*. Note that a randomly chosen half of items in the buffer is simply discarded, whereas the other half of items in the buffer is "output" by the compaction operation.

The overall KLL sketch is built as a sequence of at most $\log_2(n)$ such compactors, such that the output stream of a compactor is treated as the input stream of the next compactor. We thus think of the compactors as arranged into *levels*, with the first one at level 0. Similar compactors were already used, e.g., in [19, 20, 1, 18], and additional ideas are needed to get the optimal space bound for additive error, of $O(1/\varepsilon)$ items stored across all compactors [16].

The compactor building block is not directly applicable to our setting for the following reasons. A first observation is that to achieve the relative error guarantee, we need to always store the $1/\varepsilon$ smallest items. This is because the relative error guarantee demands that estimated ranks for the $1/\varepsilon$ lowest-ranked items in the data stream are *exact*. If even a single one of these items is deleted from the summary, then these estimates will not be exact. Similarly, among the next $2/\varepsilon$ smallest items, the summary must store essentially every other item to achieve multiplicative error. Among the next $4/\varepsilon$ smallest items in the order, the sketch must store roughly every fourth item, and so on.

The following simple modification of the compactor from the KLL sketch indeed achieves the above. Each buffer of size B "protects" the $B/2$ smallest items stored inside, meaning that these items are not involved in any compaction (i.e., the compaction operation only removes the $B/2$ largest items from the buffer). Unfortunately, it turns out that this simple approach requires space $\Theta(\varepsilon^{-2} \cdot \log(\varepsilon^2 n))$, which merely matches the space bound achieved in [29], and in particular has a suboptimal dependence on $1/\varepsilon$.

The key technical contribution of our work is to enhance this simple approach with a more sophisticated rule for selecting the number of protected items in each compaction. In this abstract, we describe a solution choosing this number in each compaction at random from an appropriate geometric distribution. In the full version [4], to get a cleaner analysis, we derandomize this distribution.

While the resulting algorithm is relatively simple, analyzing the error behavior brings new challenges that do not arise in the additive error setting. Roughly speaking, when analyzing the accuracy of the estimate for $\mathrm{R}(y)$ for any fixed item y, all error can be "attributed" to compaction operations. In the additive error setting, one may suppose that *every* compaction operation contributes to the error and still obtain a tight error analysis [16]. Unfortunately, this is not at all the case for relative error: as already indicated, to obtain our accuracy bounds it is essential to show that the estimate for any low-ranked item y is affected by very few compaction operations. Thus, the first step of our analysis is to carefully bound the number of compactions on each level that affect the error for y, using the definition of the geometric distribution in the compaction operation.

Organization of the abstract. We describe our sketch in Section 2; as mentioned above, the algorithm outlined in this abstract is slightly different to the one in the full version [4]. In Section 3, we sketch an analysis of this algorithm in the streaming setting, assuming a foreknowledge of (a polynomial upper bound on) the stream length. Finally, in Section 4 we briefly outline how to analyze the algorithm under merge operations and without any advance knowledge about the stream length as well as adjustments from [7] to make it more efficient in practice. The details can be found in the full version [4].

1.2 Detailed Comparison to Prior Work

Some prior works on streaming quantiles consider queries to be *ranks* $r \in \{1, \ldots, n\}$, and the algorithm must identify an item $y \in \mathcal{U}$ such that $\mathrm{R}(y)$ is close to r; this is called the *quantile query*. In this work, we focus on the dual problem of *rank queries*, where we consider queries to be universe items $y \in \mathcal{U}$ and the algorithm must yield an accurate estimate for $\mathrm{R}(y)$. Unless specified otherwise, algorithms described in this section directly solve both formulations (this holds for our algorithm as well). Algorithms are randomized unless stated otherwise. For simplicity, randomized algorithms are assumed to have constant failure probability δ. All reported space costs refer to the number of universe items stored. (Apart from storing universe items, the algorithms may store, for example, bounds on ranks of stored items or some counters, but the number of such variables is proportional to the number of items stored or even smaller. Thus, the space bounds are in memory words, which can store any item or an integer with $O(\log(n + |\mathcal{U}|))$ bits.)

Additive error. Manku, Rajagopalan, and Lindsay [19, 20] built on the work of Munro and Paterson [22] and gave a deterministic solution storing at most $O(\varepsilon^{-1} \cdot \log^2(\varepsilon n))$ items, assuming prior knowledge of n. Greenwald and Khanna [13] created an intricate deterministic streaming algorithm that stores $O(\varepsilon^{-1} \cdot \log(\varepsilon n))$ items. This is the best known deterministic algorithm for this problem, with a matching lower bound for comparison-based streaming algorithms [8]. Agarwal et al. [1] provided a mergeable sketch of size $O(\varepsilon^{-1} \cdot \log^{1.5}(1/\varepsilon))$. This paper contains many ideas and observations that were used in later work. Felber and Ostrovsky [11] managed to reduce the space complexity to $O(\varepsilon^{-1} \cdot \log(1/\varepsilon))$ items by combining sampling with the Greenwald-Khanna sketches in non-trivial ways. Finally, Karnin, Lang, and Liberty [16] resolved the problem by providing an $O(1/\varepsilon)$-space solution, which is optimal. For general (non-constant) failure probabilities δ, the space upper bound becomes $O(\varepsilon^{-1} \cdot \log\log(1/\delta))$, and they also prove a matching lower bound for comparison-based randomized algorithms, assuming $\delta \leq 1/n!$ (i.e., δ is exponentially small in n).

Multiplicative error. A large number of works sought to provide more accurate quantile estimates for low or high ranks. Only a handful offer solutions to the relative error quantiles problem considered in this work (sometimes also called the biased quantiles problem). Gupta and Zane [14] gave an algorithm for relative error quantiles that stores $O(\varepsilon^{-3} \cdot \log^2(\varepsilon n))$ items, and used this to approximately count the number of inversions in a list; their algorithm requires prior knowledge of the stream length n. As previously mentioned, Zhang et al. [29] presented an algorithm storing $O(\varepsilon^{-2} \cdot \log(\varepsilon^2 n))$ universe items. Cormode et al. [6] designed a deterministic sketch storing $O(\varepsilon^{-1} \cdot \log(\varepsilon n) \cdot \log |\mathcal{U}|)$ items, which requires prior knowledge of the data universe $\mathcal{U}$. Their algorithm is inspired by the work of Shrivastava et al. [26] in the additive error setting and it is also mergeable (see [1, Section 3]). Zhang and Wang [28] gave a deterministic merge-and-prune algorithm storing $O(\varepsilon^{-1} \cdot \log^3(\varepsilon n))$ items, which can handle arbitrary merges with an upper bound on n, and streaming updates for unknown n. However, it does not tackle the most general case of merging without a prior

bound on n. Cormode and Veselý [8] recently showed a space lower bound of $\Omega(\varepsilon^{-1} \cdot \log^2(\varepsilon n))$ items for any deterministic comparison-based algorithm.

Other related works that do not fully solve the relative error quantiles problem are as follows. Manku, Rajagopalan, and Lindsay [20] designed an algorithm that, for a specified number $\phi \in [0,1]$, stores $O(\varepsilon^{-1} \cdot \log(1/\delta))$ items and can return an item y with $\mathrm{R}(y)/n \in [(1-\varepsilon)\phi, (1+\varepsilon)\phi]$ (their algorithm requires prior knowledge of n). Cormode et al. [5] gave a deterministic algorithm that is meant to achieve error properties "in between" additive and relative error guarantees. That is, their algorithm aims to provide multiplicative guarantees only up to some minimum rank k; for items of rank below k, their solution only provides additive guarantees. Their algorithm does not solve the relative error quantiles problem: [29] observed that for adversarial item ordering, the algorithm of [5] requires linear space to achieve relative error for all ranks.

Dunning and Ertl [10, 9] describe a heuristic algorithm called t-digest that is intended to achieve relative error, but they provide no formal accuracy analysis. Indeed, Cormode et al. [7] show that the error of t-digest may be arbitrarily large on adversarially generated inputs. This latter paper also compares t-digest and ReqSketch (i.e., the algorithm of Theorem 1) on randomly generated inputs and proposes implementation improvements for ReqSketch that make it process an input stream faster than t-digest; see Section 4.

Most recently, Masson, Rim, and Lee [21] introduced a new notion of error for quantile sketches (they also refer to their notion as "relative error", but it is quite distinct from the notion considered in this work). They require that for a query percentile $\phi \in [0,1]$, if y denotes the item in the data stream satisfying $\mathrm{R}(y) = \phi n$, then the algorithm should return an item $\hat{y} \in \mathcal{U}$ such that $|y - \hat{y}| \leq \varepsilon \cdot |y|$. This definition only makes sense for data universes with a notion of magnitude and distance (e.g., numerical data), and the definition is not invariant to natural data transformations, such as incrementing every data item y by a large constant. It is also trivially achieved by maintaining a (mergeable) histogram with buckets $((1+\varepsilon)^i, (1+\varepsilon)^{i+1}]$. In contrast, the standard notion of relative error considered in this work does not refer to the data items themselves, only to their ranks, and is arguably of more general applicability.

2. DESCRIPTION OF THE ALGORITHM

2.1 The Relative-Compactor Object

The crux of our algorithm is a building block that we call the relative-compactor. Roughly speaking, this object processes a stream of n items and outputs a stream of at most $n/2$ items (each "up-weighted" by a factor of 2), meant to "approximate" the input stream. It does so by maintaining a buffer of limited capacity.

Our complete sketch, described in Section 2.2 below, is composed of a sequence of relative-compactors, where the input of the $(h+1)$'th relative-compactor is the output of the h'th. With at most $\log_2(\varepsilon n)$ such relative-compactors, n being the length of the input stream, the output of the last relative-compactor is of size $O(1/\varepsilon)$, and hence can be stored in memory.

Compaction operations. The basic subroutine used by our relative-compactor is a compaction operation. The input to a compaction operation is a list X of $2m$ items $x_1 \leq x_2 \leq \ldots \leq x_{2m}$, and the output is a sequence Z of m items. This output is chosen to be one of the following two sequences, uniformly at random: Either $Z = \{x_{2i-1}\}_{i=1}^{m}$ or $Z = \{x_{2i}\}_{i=1}^{m}$. That is, the output sequence Z equals either the even or odd indexed items in the sorted order of X, with both outcomes equally probable.

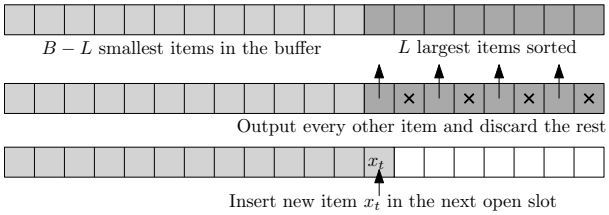


Figure 1: Illustration of the execution of a relative-compactor when inserting a new item x_t into a buffer that is full at time t. See lines 5-13 of Algorithm 1.

Consider an item $y \in \mathcal{U}$ and recall that $\mathrm{R}(y; X) = |\{x \in X \mid x \leq y\}|$ is the number of items $x \in X$ satisfying $x \leq y$ (we remark that both X and $\{x \in X \mid x \leq y\}$ are multisets of universe items). The following is a trivial observation regarding the error of the rank estimate of y with respect to the input X of a compaction operation when using Z. We view the output Z of a compaction operation (with all items up-weighted by a factor of 2) as an approximation to the input X; for any y, its weighted rank in Z should be close to its rank in X. Observation 2.1 below states that this approximation incurs *zero error* on items that have an even rank in X. Moreover, for items y that have an odd rank in X, the error for $y \in \mathcal{U}$ introduced by the compaction operation is $+1$ or -1 with equal probability.

OBSERVATION 2.1. *A universe item $y \in \mathcal{U}$ is said to be even (odd) w.r.t a compaction operation if $\mathrm{R}(y; X)$ is even (odd), where X is the input sequence to the operation. If y is even w.r.t the compaction, then $\mathrm{R}(y; X) - 2\,\mathrm{R}(y; Z) = 0$. Otherwise, $\mathrm{R}(y; X) - 2\,\mathrm{R}(y; Z)$ is a variable taking a value from $\{-1, 1\}$ uniformly at random.*

The observation that items of even rank (and in particular items of rank zero) suffer no error from a compaction operation plays an especially important role in the error analysis of our full sketch.

Full description of the relative-compactor. The complete description of the relative-compactor object is given in Algorithm 1. The high-level idea is as follows. The relative-compactor maintains a buffer of size $B = 2 \cdot k \cdot \lceil \log_2(n/k) \rceil$ where k is an even integer parameter controlling the error and n is the upper bound on the stream length. (In this abstract, we assume that such an upper bound is available; we discuss removing this assumption in Section 4.) The incoming items are stored in the buffer until it is full, and then we perform a compaction operation, as described above.

The input to the compaction operation is not all items in the buffer, but rather the largest L items in the buffer for a parameter $L \leq B/2$ such that L is even. These L largest items are then removed from the buffer, and the output of the compaction operation is sent to the output stream of the buffer. This intuitively lets low-ranked items stay in the buffer longer than high-ranked ones. Indeed, the lowest-

Figure 2: Illustration of a relative-compactor and its sections, together with the indexes of the sections.

Algorithm 1 Relative-Compactor

Input: Parameters $k \in 2\mathbb{N}^+$ and $n \in \mathbb{N}^+$, and a stream of items $x_1, x_2, \ldots$ of length at most n

Set $m = \lceil \log_2(n/k) \rceil$ ▷ Number of sections
Set $B = 2 \cdot k \cdot m$ ▷ Buffer size
Initialize an empty buffer $\mathcal{B}$ of size B, indexed from 1
for $t = 1 \ldots$ **do**
  if $\mathcal{B}$ is full **then** ▷ Compaction operation
    Randomly pick section $I \in \{0, \ldots, m-1\}$ with $\Pr[I = i] = p_i := 2^i/(2^m - 1)$
    Set $L = i \cdot k$ and $S = B - L + 1$
    Pivot $\mathcal{B}$ s.t. the largest L items occupy $\mathcal{B}[S : B]$
      ▷ $\mathcal{B}[S : B]$ are the last L slots of $\mathcal{B}$
    Sort $\mathcal{B}[S : B]$ in non-descending order
    Output either even or odd indexed items in the range $\mathcal{B}[S : B]$ with equal probability
    Mark slots $\mathcal{B}[S : B]$ in the buffer as clear
  Store x_t into the next available slot in the buffer $\mathcal{B}$.

Algorithm 2 ReqSketch (Relative-Error Quantiles sketch)

Input: Parameters $k \in 2\mathbb{N}^+$ and $n \in \mathbb{N}^+$, and a stream of items $x_1, x_2, \ldots$ of length at most n
Output: A sketch answering rank (and quantile) queries

Let RelCompactors be a list of relative-compactors
Set $H = 0$ and initialize relative-compactor with parameters k and n at RelCompactors[0]
for $t = 1 \ldots$ **do**
  INSERT(x_t, 0)
function INSERT(x,h)
  if $H < h$ **then**
    Set $H = h$ and initialize relative-compactor with parameters k and n at RelCompactors[h]
  Insert item x into RelCompactors[h]
  for z in output stream of RelCompactors[h] **do**
    INSERT($z, h+1$) ▷ Items output by the compaction (if any)
function ESTIMATE-RANK(y)
  Set $\hat{\mathrm{R}}(y) = 0$
  for $h = 0$ to H **do**
    for each item $y' \le y$ in RelCompactors[h] **do**
      Increment $\hat{\mathrm{R}}(y)$ by 2^h
  return $\hat{\mathrm{R}}(y)$

ranked half of items in the buffer are *never* removed. We show later that this facilitates the relative error guarantee.

The crucial part in the design of Algorithm 1 is to select the parameter L in a right way, as L controls the number of items compacted each time the buffer is full. If we were to set $L = B/2$ for all compaction operations, then analyzing the worst-case behavior reveals that we need $B \approx 1/\varepsilon^2$, resulting in a sketch with a quadratic dependency on $1/\varepsilon$. To achieve the linear dependency on $1/\varepsilon$, we choose the parameter L via a suitable geometric distribution subject to the constraint that $L \le B/2$.

In more detail, during each compaction operation, the second half of the buffer (with $B/2$ largest items) is split into $m := \lceil \log_2(n/k) \rceil$ sections, each of size k and indexed $0, \ldots, m-1$ from the left; see Figure 2. The first section involved in the compaction is selected with probability exponentially increasing with its index, namely, section $i \in \{0, \ldots, m-1\}$ is chosen with probability $p_i := 2^i \cdot \gamma$, where $\gamma = 1/(2^m - 1)$ is chosen so that the probabilities p_i sum to 1. Furthermore, the chosen geometric distribution has the following property:

$$p_i = \sum_{j=0}^{i-1} p_j + \gamma\,. \tag{1}$$

That is, the probability of starting the compaction in section i is essentially equal to the probability of also compacting section $i-1$. Since $B = 2 \cdot k \cdot m$ and since at most $m \cdot k$ largest items are involved in the compaction, the smallest $B/2$ items in the buffer are never removed.

REMARK 2. *In the original version of the paper, we describe the algorithm with a* derandomized *geometric distribution. Thus, that version of the algorithm uses randomness only to select which items to place in the output stream, not how many items to compact. This leads to a cleaner analysis and isolates the one component of the algorithm for which randomness is essential. In this abstract, we have chosen not to derandomize the geometric distribution for simplicity.*

2.2 The Full Sketch

Following prior work [19, 1, 16], the full sketch uses a sequence of relative-compactors. At the very start of the stream, it consists of a single relative-compactor (at level 0) and opens a new one (at level 1) once items are fed to the output stream of the first relative-compactor (i.e., after the first compaction operation, which occurs on the first stream update during which the buffer is full). In general, when the newest relative-compactor is at level h, the first time the buffer at level h performs a compaction operation (feeding items into its output stream for the first time), we open a new relative-compactor at level $h+1$ and feed it these items. Algorithm 2 describes the logic of this sketch.

To answer rank queries, we use the items in the buffers of the relative-compactors as a weighted coreset. That is, the union of these items is a weighted set $\mathcal{C}$ of items, where the weight of items in relative-compactor at level h is 2^h (recall that h starts from 0), and the approximate rank of y, denoted $\hat{\mathrm{R}}(y)$, is the sum of weights of items in $\mathcal{C}$ smaller than or equal to y. Similarly, ReqSketch can answer quantile queries, i.e., for a given rank $r \in \{1, \ldots, n\}$, return an item $y \in \mathcal{U}$ with $\mathrm{R}(y)$ close to r; the algorithm just returns an item y stored in one of the relative-compactors with $\hat{\mathrm{R}}(y)$ closest to the query rank r among all items in the sketch.

The construction of layered exponentially-weighted compactors and the subsequent rank estimation is virtually identical to that explained in prior works [19, 1, 16]. Our essential departure from prior work is in the definition of the compaction operation, not in how compactors are plumbed together to form a complete sketch.

Merge operation. The merge operation takes as input two sketches S' and S'' which have processed two separate streams

σ' and σ'' and outputs a sketch S summarizing the concatenated stream $\sigma = \sigma' \circ \sigma''$ (the order of σ' and σ'' does not matter here). For the simplified sketch presented in this abstract, merging two sketches is straightforward: At each level, concatenate the buffers and if that causes the capacity of the compactor to be reached or exceeded, perform the compaction operation, as in Algorithm 1. There are additional complications when an upper bound n on the combined input size is not available in advance; for instance, the number of sections m (and the section size k) may need to be adjusted during a merge operation. These details are described in the full version [4].

3. ANALYSIS

We provide a sketch of the analysis in the streaming setting, assuming a foreknowledge of (an upper bound on) the stream length n. To analyze the error of the full sketch, we focus on the error in the estimated rank of an arbitrary fixed item $y \in \mathcal{U}$. Let $\mathrm{R}(y)$ be the rank of item y in the input stream, and let $\mathrm{Err}(y) = \hat{\mathrm{R}}(y) - \mathrm{R}(y)$ be the error of the estimated rank for y.

Analysis of the relatively-compactor. We first restrict our attention to any single relative-compactor at level h (Algorithm 1) maintained by our sketching algorithm (Algorithm 2), and we use "time t" to refer to the t'th insertion operation to this particular relative-compactor. We analyze the error introduced by the relative-compactor for item y. Specifically, at time t, let $X^t = (x_1, \ldots, x_t)$ be the prefix of the input stream to the relative-compactor, Z^t be the output stream, and $\mathcal{B}^t$ be the items in the buffer after inserting item x_t. The rank error made by the relative-compactor at time t with respect to item y is defined as

$$\mathrm{Err}_h^t(y) = \mathrm{R}(y; X^t) - 2\,\mathrm{R}(y; Z^t) - \mathrm{R}(y; \mathcal{B}^t). \tag{2}$$

Conceptually, $\mathrm{Err}_h^t(y)$ tracks the difference between y's rank in the input stream X^t at time t versus its rank as estimated by the combination of the output stream and the remaining items in the buffer at time t (output items are upweighted by a factor of 2 while items remaining in the buffer are not). We denote the overall error of the relative-compactor by $\mathrm{Err}_h(y)$ and the total number of items $x \le y$ inserted to the level-h buffer by $\mathrm{R}_h(y)$. Then $\mathrm{Err}_h(y) = \mathrm{R}_h(y) - 2\,\mathrm{R}_{h+1}(y) - \mathrm{R}(y; \mathcal{B}_h)$, where $\mathcal{B}_h$ is the level-h buffer after Algorithm 2 has processed the input stream (note that $\mathrm{R}_{h+1}(y)$ is the number of items $x \le y$ in the output stream of the level-h relative-compactor). To bound $\mathrm{Err}_h(y)$, we keep track of the error associated with y over time, and define the increment or decrement of it as

$$\Delta_h^t(y) = \mathrm{Err}_h^t(y) - \mathrm{Err}_h^{t-1}(y),$$

where $\mathrm{Err}_h^0(y) = 0$.

Clearly, if the algorithm performs no compaction operation in a time step t, then $\Delta_h^t(y) = 0$. (Recall that a compaction is an execution of lines 6-12 of Algorithm 1.) Let us consider what happens in a step t in which a compaction operation occurs. Observation 2.1 shows that if y is even with respect to the compaction, then y suffers no error, meaning that $\Delta_h^t(y) = 0$. Otherwise, $\Delta_h^t(y)$ is uniform in $\{-1, 1\}$.

It follows that $\mathbb{E}[\mathrm{Err}_h(y)] = 0$, which implies that the estimator $\hat{\mathrm{R}}(y)$ is unbiased. To bound the variance of $\mathrm{Err}_h(y)$, we analyze $\mathrm{Var}[\Delta_h^t(y)]$ for any step t with a compaction operation. We suppose that $\mathrm{R}(y; \mathcal{B}^t) > B/2$ as otherwise, no item $x \le y$ is removed from the buffer and $\Delta_h^t(y) = 0$. Below, we only focus on steps t with a compaction operation such that $\mathrm{R}(y; \mathcal{B}^t) > B/2$. Let $i_t \in \{0, \ldots, m-1\}$ be the largest index of a section of $\mathcal{B}^t$ with an item $x \le y$, i.e., i_t is the smallest integer $i \ge 0$ such that $\mathrm{R}(y; \mathcal{B}^t) \le B/2 + (i+1) \cdot k$. Note that if Algorithm 1 draws $I < i_t$ in line 6 during the compaction in step t, at least k items $x \le y$ are removed from the buffer, which implies that the number of steps t with $I < i_t$ is at most $\mathrm{R}_h(y)/k$. Using this observation, our aim is to bound $\mathrm{Var}[\mathrm{Err}_h(y)]$ in terms of $\mathrm{R}_h(y)/k$.

The probability of $\Delta_h^t(y) \neq 0$ is at most the probability of removing an item $x \le y$, which by the definition of i_t and Algorithm 1 equals

$$\sum_{j=0}^{i_t} p_j = \sum_{j=0}^{i_t-1} p_j + p_{i_t} = 2 \sum_{j=0}^{i_t-1} p_j + \gamma\,, \tag{3}$$

where we use (1); this is where we rely on using the geometric distribution to choose the number of sections involved in the compaction (recall that $\gamma = 1/(2^m - 1)$ is small). Using (3) and that $\Delta_h^t(y) \in \{-1, 0, 1\}$, we get that $\mathrm{Var}[\Delta_h^t(y)] = \mathbb{E}[\big(\Delta_h^t(y)\big)^2] = \Pr[\Delta_h^t(y) \neq 0] \le 2 \sum_{i=0}^{i_t-1} p_i + \gamma$.

Since the variables $\Delta_h^t(y)$ are independent,

$$\mathrm{Var}[\mathrm{Err}_h(y)] = \sum_t \mathrm{Var}[\Delta_h^t(y)] = \sum_t \left(2 \sum_{i=0}^{i_t-1} p_i + \gamma \right).$$

As observed above, the number of steps t with $I < i_t$ in line 6 is at most $\mathrm{R}_h(y)/k$, which implies that $\sum_t \sum_{i=0}^{i_t-1} p_i \le \mathrm{R}_h(y)/k$ (we omit a formal proof of this observation).

As a compaction is performed at most once in every k steps, we have that $\sum_t \gamma \le \gamma \cdot n/k \le n/(n-k) \le 2$, using the definition of $\gamma = 1/(2^m - 1)$ and $m = \lceil \log_2(n/k) \rceil$; the last step uses $n \ge 2k$ as otherwise the whole input to the relative-compactor would be stored in the buffer which has size at least $2k$ and there would be no compaction operation.

Summarizing, we have obtained the following bound on the variance of the error at level h:

$$\mathrm{Var}[\mathrm{Err}_h(y)] \le \frac{2\,\mathrm{R}_h(y)}{k} + 2 \le \frac{4\,\mathrm{R}_h(y)}{k}\,, \tag{4}$$

where the second inequality uses $\mathrm{R}_h(y) \ge k$ as otherwise there would be no level-h compaction operation affecting the error for y, i.e., $\mathrm{Err}_h(y) = 0$ if $\mathrm{R}_h(y) < k$

Analysis of the full sketch in the streaming setting. To make the variance bound for a single level in (4) useful, we show that $\mathrm{R}_h(y)$ roughly halves with every level. This is easy to see in expectation, and it holds with high probability up to a certain crucial level $H(y)$. Here, we define $H(y)$ to be the minimal h for which $2^{-h+1}\,\mathrm{R}(y) \le B/2$. For $h = H(y) - 1$ (assuming $H(y) > 0$), we particularly have $2^{2-H(y)}\,\mathrm{R}(y) > B/2$, or equivalently, $2^{H(y)} < 2^3 \cdot \mathrm{R}(y)/B$.

LEMMA 3.1. *With probability at least $1 - \delta$ it holds that $\mathrm{R}_h(y) \le 2^{-h+1}\,\mathrm{R}(y)$ for any $h \le H(y)$.*

We omit the proof by induction, using Chernoff bounds (which also requires choosing k as described below). In what follows, we condition on the bound on $\mathrm{R}_h(y)$ in Lemma 3.1 for any $h \le H(y)$. For $h = H(y)$, we thus have that $\mathrm{R}_{H(y)}(y) \le 2^{-H(y)+1}\,\mathrm{R}(y) \le B/2$ and that $\mathrm{R}_h(y) = 0$ for any $h > H(y)$. Thus, Observation 2.1 implies that $\mathrm{Err}_h(y) = 0$ for any $h \ge H(y)$.

We are now ready to bound the overall error of the sketch for item y, i.e., $\mathrm{Err}(y) = \hat{\mathrm{R}}(y) - \mathrm{R}(y)$ where $\hat{\mathrm{R}}(y)$ is the estimated rank of y. By the observations above, we get

$$\mathrm{Err}(y) = \sum_{h=0}^{H(y)-1} 2^h \,\mathrm{Err}_h(y)\,.$$

Recall that a zero-mean random variable X with variance σ^2 is sub-Gaussian if $\mathbb{E}[\exp(sX)] \le \exp(-\frac{1}{2}\cdot s^2\cdot\sigma^2)$ for any $s\in\mathbb{R}$; note that a weighted sum of independent zero-mean sub-Gaussian variables is a zero-mean sub-Gaussian random variable as well. By the analysis of level h, $\mathrm{Err}_h(y)$ is a zero-mean sub-Gaussian variable with $\mathrm{Var}[\mathrm{Err}_h(y)] \le 4\,\mathrm{R}_h(y)/k$. It follows that $\mathrm{Err}(y)$ is a zero-mean sub-Gaussian random variable with variance

$$\begin{aligned}\sum_{h=0}^{H(y)-1} 2^{2h}\,\mathrm{Var}[\mathrm{Err}_h(y)] &\le \sum_{h=0}^{H(y)-1} 2^{2h}\cdot\frac{4\,\mathrm{R}_h(y)}{k}\\ &\le \sum_{h=0}^{H(y)-1} 2^{h+3}\cdot\frac{\mathrm{R}(y)}{k}\\ &\le 2^{H(y)+3}\cdot\frac{\mathrm{R}(y)}{k} \le 2^6\cdot\frac{\mathrm{R}(y)^2}{k\cdot B}\,,\end{aligned}$$

where the second inequality is due to Lemma 3.1 and the last inequality follows from the definition of $H(y)$.

Given the desired accuracy ε and the desired upper bound δ on failure probability, we choose k so that $\mathrm{Var}[\mathrm{Err}(y)] \le \varepsilon^2\,\mathrm{R}(y)^2/\ln(1/\delta)$, or equivalently that $k\cdot B = k\cdot 2\cdot k\cdot\lceil\log_2(n/k)\rceil \ge \Theta\left(\varepsilon^{-2}\cdot\ln(1/\delta)\right)$; this holds for k satisfying $k = \Theta\left(\varepsilon^{-1}\cdot\sqrt{\ln\frac{1}{\delta}/\log_2(n/k)}\right)$. Then using standard (Chernoff) tail bounds for sub-Gaussian variables concludes the calculation of the failure probability.

Finally, for the space bound, by the choice of k above we have that $B = O\left(\varepsilon^{-1}\cdot\sqrt{\log(\varepsilon n)\cdot\log(1/\delta)}\right)$, and it thus remains to observe that the number of relative-compactors ever created by Algorithm 2 is at most $O(\log(\varepsilon n))$. Indeed, each item on level h has weight 2^h, so there are at most $n/2^h$ items inserted to the buffer at that level. For $h = \lceil\log_2(n/B)\rceil$, we get that on this level, there are fewer than B items inserted to the buffer, which is consequently not compacted, so the highest level has index at most $\lceil\log_2(n/B)\rceil$. (In the case $\varepsilon > \sqrt{\ln\frac{1}{\delta}/\log_2(\varepsilon n)}$, we have that $k = O(1)$ and $B = O(\log(\varepsilon n)$.)

4. ANALYSIS EXTENSIONS

We discuss how to extend the analysis presented above to more general settings. We also outline important practical optimizations that are included in the ReqSketch implementation in the Apache DataSketches library [25].

Handling unknown stream lengths. In previous sections, we outlined a proof of Theorem 1 in the streaming setting assuming that (an upper bound on) n is known, where n is the true stream length. The space usage of the algorithm grows polynomially with the logarithm of this upper bound, so if this upper bound is at most n^c for some constant $c \ge 1$, then the space usage of the algorithm will remain as stated in Theorem 1, up to a constant factor.

In the case that such a polynomial upper bound on n is not known, we modify the algorithm slightly, and start with an initial estimate N_0 of n, namely, $N_0 = \Theta(\varepsilon^{-1})$. That is, we begin by running Algorithm 2 with parameters k and N_0. As soon as the stream length hits the current estimate N_i, the algorithm "closes out" the current data structure and continues to store it in "read only" mode, while initializing a new summary based on the estimated stream length of $N_{i+1} = N_i^2$ (i.e., we execute Algorithm 2 with parameters k and N_{i+1}). This process occurs at most $\log_2\log_2(\varepsilon n)$ many times, before the guess is at least the true stream length n. At the end of the stream, the rank of any item y is estimated by summing the estimates returned by each of the at most $\log_2\log_2(\varepsilon n)$ summaries stored by the algorithm. A simple extension of the analysis presented above shows that the accuracy-space tradeoff from Theorem 1 holds for this algorithm. Nevertheless, in a practical implementation, we suggest not to close out the current summary and only recompute the parameters of each buffer as described below.

Full mergeability. There are substantial additional technical difficulties to analyze the algorithm under an arbitrary sequence of merge operations, especially with no foreknowledge of the total size of the input. Most notably, when the input size is not known in advance, the parameters of k and B of each relative-compactor must change as more inputs are processed. This makes obtaining a tight bound on the variance of the resulting estimates highly involved. In particular, as a sketch processes more and more inputs, it protects more and more items, which means that items appearing early in the stream may *not* be protected by the sketch, even though they *would have been protected* if they appeared later in the stream (this is because the buffer size increases as the summarized input size grows). As mentioned above, addressing this issue is reasonably simple in the streaming setting. However, that simple approach does not work for a general sequence of merge operations, and the full version [4] contains a much more intricate analysis to give a fully mergeable summary.

Practical adjustments of ReqSketch. The description of our algorithm in Section 2.2 is suitable for a mathematical analysis, however, there are several ways how to improve its efficiency, which are described in [7]. First, we observe that the adjustments of the KLL sketch proposed by Ivkin et al. [15] are applicable to ReqSketch as well, for example, we can allow the buffer to exceed its capacity provided that the overall capacity of the sketch is satisfied; this can be viewed as laziness in performing compaction operations.

The main difference between the KLL sketch and ReqSketch is the use of the geometric distribution to choose the number of compacted items, and this requires the buffer size to depend on the input size n. Instead of setting the buffer size based on an upper bound on n, it is sufficient to count the number C_h of compaction operations at each level h and set the level-h buffer size to $O(\varepsilon^{-1}\cdot\sqrt{\log C_h})$, recomputing the buffer size only when $\log_2 C_h$ increases by a factor of 2; then we also decrease the section size k by a factor of $\sqrt{2}$. With this adjustment, the buffer size will be *non-increasing* in the level h. More importantly, Cormode et al. [7] observed that the aforementioned laziness proposed in [15] should be restricted to level 0 only (which has the largest size). In other words, when we compact level 0, we perform the compaction operation at any other level in which the buffer exceeds its capacity. Somewhat surprisingly, this "partial laziness" significantly decreases the observed update time (averaged over processing the whole stream), compared to the original proposal from [15].

5. CONCLUSIONS

For constant failure probability δ, we show an $O(\varepsilon^{-1} \cdot \log^{1.5}(\varepsilon n))$ space upper bound for relative error quantile approximation over data streams. Our algorithm is provably more space-efficient than any deterministic comparison-based algorithm, and is within an $O(\sqrt{\log(\varepsilon n)})$ factor of the known lower bound for randomized algorithms (even non-streaming algorithms). The sketch output by our algorithm is fully mergeable, with the same accuracy-space trade-off as in the streaming setting, rendering it suitable for a parallel or distributed environment. The main open question is to close the aforementioned $O(\sqrt{\log(\varepsilon n)})$-factor gap.

6. ACKNOWLEDGMENTS

The research is performed in close collaboration with DataSketches [25], the Apache open source project for streaming data analytics. G. Cormode and P. Veselý were supported by European Research Council grant ERC-2014-CoG 647557. P. Veselý was also partially supported by the project 19-27871X of GA ČR and by Charles University project UNCE/SCI/004. J. Thaler was supported by NSF SPX award CCF-1918989 and NSF CAREER award CCF-1845125.

7. REFERENCES

[1] P. K. Agarwal, G. Cormode, Z. Huang, J. M. Phillips, Z. Wei, and K. Yi. Mergeable summaries. *ACM Transactions on Database Systems*, 38(4):26, 2013.

[2] R. Agrawal and A. Swami. A one-pass space-efficient algorithm for finding quantiles. In *COMAD-95, Pune, India*, 1995.

[3] A. Arasu and G. S. Manku. Approximate counts and quantiles over sliding windows. In *PODS '04*, pages 286–296. ACM, 2004.

[4] G. Cormode, Z. Karnin, E. Liberty, J. Thaler, and P. Veselý. Relative error streaming quantiles. *arXiv preprint arXiv:2004.01668*, 2020.

[5] G. Cormode, F. Korn, S. Muthukrishnan, and D. Srivastava. Effective computation of biased quantiles over data streams. In *ICDE '05*, pages 20–31, Washington, DC, USA, 2005. IEEE Computer Society.

[6] G. Cormode, F. Korn, S. Muthukrishnan, and D. Srivastava. Space- and time-efficient deterministic algorithms for biased quantiles over data streams. In *PODS '06*, pages 263–272. ACM, 2006.

[7] G. Cormode, A. Mishra, J. Ross, and P. Veselý. Theory meets practice at the median: A worst case comparison of relative error quantile algorithms. In *KDD '21*, page 2722–2731, New York, NY, USA, 2021. ACM.

[8] G. Cormode and P. Veselý. A tight lower bound for comparison-based quantile summaries. In *PODS '20*, pages 81–93, New York, NY, USA, 2020. ACM.

[9] T. Dunning. The t-digest: Efficient estimates of distributions. *Software Impacts*, 7:100049, 2021.

[10] T. Dunning and O. Ertl. Computing extremely accurate quantiles using t-digests. *CoRR*, abs/1902.04023, 2019.

[11] D. Felber and R. Ostrovsky. A randomized online quantile summary in O(1/epsilon * log(1/epsilon)) words. In *APPROX/RANDOM '15*, volume 40 of *LIPIcs*, pages 775–785, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[12] S. Ganguly. A nearly optimal and deterministic summary structure for update data streams. *arXiv preprint cs/0701020*, 2007.

[13] M. Greenwald and S. Khanna. Space-efficient online computation of quantile summaries. In *ACM SIGMOD Record*, volume 30, pages 58–66. ACM, 2001.

[14] A. Gupta and F. X. Zane. Counting inversions in lists. In *SODA '03*, pages 253–254, Philadelphia, PA, USA, 2003. SIAM.

[15] N. Ivkin, E. Liberty, K. Lang, Z. Karnin, and V. Braverman. Streaming quantiles algorithms with small space and update time. *arXiv preprint arXiv:1907.00236*, 2019.

[16] Z. Karnin, K. Lang, and E. Liberty. Optimal quantile approximation in streams. In *FOCS '16*, pages 71–78. IEEE, 2016.

[17] E. Liberty and P. Veselý. relativeErrorSketch.py. In `https://github.com/edoliberty/streaming-quantiles/`, 2021.

[18] G. Luo, L. Wang, K. Yi, and G. Cormode. Quantiles over data streams: Experimental comparisons, new analyses, and further improvements. *The VLDB Journal*, 25(4):449–472, Aug. 2016.

[19] G. S. Manku, S. Rajagopalan, and B. G. Lindsay. Approximate medians and other quantiles in one pass and with limited memory. In *ACM SIGMOD Record*, volume 27, pages 426–435. ACM, 1998.

[20] G. S. Manku, S. Rajagopalan, and B. G. Lindsay. Random sampling techniques for space efficient online computation of order statistics of large datasets. In *ACM SIGMOD Record*, volume 28, pages 251–262. ACM, 1999.

[21] C. Masson, J. E. Rim, and H. K. Lee. Ddsketch: A fast and fully-mergeable quantile sketch with relative-error guarantees. *PVLDB*, 12(12):2195–2205, 2019.

[22] J. I. Munro and M. S. Paterson. Selection and sorting with limited storage. *Theoretical computer science*, 12(3):315–323, 1980.

[23] I. Pohl. *A minimum storage algorithm for computing the median*. IBM TJ Watson Research Center, 1969.

[24] V. Poosala, V. Ganti, and Y. E. Ioannidis. Approximate query answering using histograms. *IEEE Data Eng. Bull.*, 22(4):5–14, 1999.

[25] L. Rhodes, K. Lang, J. Malkin, A. Saydakov, E. Liberty, and J. Thaler. DataSketches: A library of stochastic streaming algorithms. Open source software: `https://datasketches.apache.org/`, 2013.

[26] N. Shrivastava, C. Buragohain, D. Agrawal, and S. Suri. Medians and beyond: new aggregation techniques for sensor networks. In *SenSys '04*, pages 239–249. ACM, 2004.

[27] G. Tene. How NOT to measure latency. `https://www.youtube.com/watch?v=lJ8ydIuPFeU`, 2015.

[28] Q. Zhang and W. Wang. An efficient algorithm for approximate biased quantile computation in data streams. In *CIKM '07*, pages 1023–1026, 2007.

[29] Y. Zhang, X. Lin, J. Xu, F. Korn, and W. Wang. Space-efficient relative error order sketch over data streams. In *ICDE '06*, pages 51–51. IEEE, 2006.

Technical Perspective: Structure and Complexity of Bag Consistency

Mihalis Yannakakis
Columbia University, USA

The paper *Structure and Complexity of Bag Consistency* by Albert Atserias and Phokion Kolaitis [1] studies fundamental structural and algorithmic questions on the global consistency of databases in the context of bag semantics. A collection D of relations is called *globally consistent* if there is a (so-called "universal") relation over all the attributes that appear in all the relations of D whose projections yield the relations in D. The basic algorithmic problem for consistency is: given a database D, determine whether D is globally consistent. An obvious necessary condition for global consistency is *local* (or *pairwise*) *consistency*: every pair of relations in D must be consistent. This condition is not sufficient however: It is possible that every pair is consistent, but there is no single global relation over all the attributes whose projections yield the relations in D. A natural structural question is: Which database schemas have the property that every locally consistent database over the schema is also globally consistent?

These questions were studied early on, as the theory of relational databases was being developed, in a model where relations are assumed to be sets, i.e. have no duplicate tuples. On the structural side, it was shown that the class of acyclic database schemas characterizes precisely those schemas for which local consistency is equivalent to global consistency [2]. On the algorithmic side, the global consistency problem was shown to be in general NP-complete [5]. If the database schema is acyclic however, the consistency problem can be solved in polynomial time.

The paper [1] by Atserias and Kolaitis addresses these basic questions in the model where the relations are bags (multisets). The relationship between the set-based and the bag-based model is by no means straightforward, and results may not necessarily carry over from one model to the other. For example, the classical problem of containment of conjunctive queries has been long well-understood in the set model (the problem is NP-complete [3]). However, the problem appears to be much harder in the bag model, and it is still not even known to be decidable; this discrepancy was pointed out in [4], which raised the issue of revisiting fundamental problems of database theory in the bag model.

Regarding the structural question of when local consistency is equivalent to global consistency, [1] shows that the same characterization holds in the bag model, i.e. the local-to-global consistency property holds precisely for the class of acyclic schemas. The proof however requires several new ideas and techniques (including linear programming and network flows) because some basic features of the set model do not hold any more in the bag model. For example, in the set model, if the database is consistent, then the join of all the relations can serve as the universal relation (i.e. its projections yield the relations of the database). This does not hold in the bag model even for two relations.

Regarding the algorithmic question of determining whether a given database is consistent, [1] provides a sharp dichotomy theorem for all fixed database schemas in the bag model: if the schema is acyclic then the problem can be solved in polynomial time, whereas if it is cyclic then the problem is NP-complete. The NP-completeness holds for example even for a schema with 3 relations and 3 attributes. Note, by contrast, that in the set model, if the schema is fixed, then the consistency problem can be solved trivially in polynomial time (even if the schema is cyclic), where the degree of the polynomial depends on the schema.

The paper addresses fundamental consistency questions in the bag model and resolves them exactly using nontrivial, elegant techniques. The relationship between local and global consistency arises in a variety of other settings, for example for probability distributions, and in quantum information. In the last section of their paper, Atserias and Kolaitis discuss a general model of relations over semirings, which includes as special cases the set and the bag model, and provide a preview of interesting forthcoming work on the local-to-global consistency problem in this general setting.

1. REFERENCES

[1] A. Atserias, P. Kolaitis. Structure and complexity of bag consistency. In *Proc. 40th ACM Symp. on Principles of Database Systems*, pp. 247-259, 2021.

[2] C. Beeri, R. Fagin, D. Maier, M. Yannakakis. On the desirability of acyclic database schemes. *Journal of ACM*, 30(3): 479-513, 1983.

[3] A. K. Chandra, P. Merlin. Optimal implementation of conjunctive queries in relational databases. In *Proc. 9th ACM Symp. on Theory of Computing*, pp. 77-90, 1977.

[4] S. Chaudhuri, M. Y. Vardi. Optimization of real conjunctive queries, In *Proc. 12th ACM Symp. on Principles of Database Systems*, pp. 59-70, 1993.

[5] P. Honeyman, R. Ladner, M. Yannakakis. Testing the universal instance assumption. *Inf. Process. Lett.*, 10(1):14-19, 1979.

Structure and Complexity of Bag Consistency

Albert Atserias
Universitat Politècnica de Catalunya
Barcelona, Catalonia
Spain
atserias@cs.upc.edu

Phokion G. Kolaitis
UC Santa Cruz and IBM Research
Santa Cruz, California
USA
kolaitis@ucsc.edu

ABSTRACT

Since the early days of relational databases, it was realized that acyclic hypergraphs give rise to database schemas with desirable structural and algorithmic properties. In a by-now classical paper, Beeri, Fagin, Maier, and Yannakakis established several different equivalent characterizations of acyclicity; in particular, they showed that the sets of attributes of a schema form an acyclic hypergraph if and only if the local-to-global consistency property for relations over that schema holds, which means that every collection of pairwise consistent relations over the schema is globally consistent. Even though real-life databases consist of bags (multisets), there has not been a study of the interplay between local consistency and global consistency for bags. We embark on such a study here and we first show that the sets of attributes of a schema form an acyclic hypergraph if and only if the local-to-global consistency property for bags over that schema holds. After this, we explore algorithmic aspects of global consistency for bags by analyzing the computational complexity of the global consistency problem for bags: given a collection of bags, are these bags globally consistent? We show that this problem is in NP, even when the schema is part of the input. We then establish the following dichotomy theorem for fixed schemas: if the schema is acyclic, then the global consistency problem for bags is solvable in polynomial time, while if the schema is cyclic, then the global consistency problem for bags is NP-complete. The latter result contrasts sharply with the state of affairs for relations, where, for each fixed schema, the global consistency problem for relations is solvable in polynomial time.

1. INTRODUCTION

This paper bring together two different strands of research in database theory: the study of global consistency and the study of bag semantics. Before presenting an overview of our main results, we provide some background to each of these two strands.

Minor revision of the paper with the same title published in the Proceedings of the 40th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS 2021), ISBN978-1-4503-8381-3/21/06, June 20-25, 2021. DOI:10.1145/3452021.3458329

The study of global consistency in relational databases arose from the universal relation model, which is the assumption that all relations at hand are projections of a single relation, called the *universal relation*. Much of the work on database dependencies and normalization during the 1970s made this assumption first implicitly and then explicitly, as for instance in the paper by Beeri, Bernstein, and Goodman [8]. The universal relation model implies that occurrences of the same attribute in different relations have the same meaning; it also provides a framework to study dependencies across different relations. Furthermore, it has been argued that the universal relation model yields logical independence and access-path independence [22], thus it can be regarded as an early model of data integration. At times, the universal relation model was surrounded by controversy with arguments both against it [18] and in favor of it [24]. The controversy notwithstanding and instead of assuming the presence of a universal relation, researchers also investigated *when* a universal relation exists.

On the algorithmic side, the universal relation problem (also known as the global consistency problem) is the following decision problem: given relations $R_1, \ldots, R_m$, is there a relation R such that, for every $i \leq m$, the projection of R on the attributes of R_i is equal to R_i? If the answer is positive, then the relations $R_1, \ldots, R_m$ are said to be *globally consistent* and R is said to be a *universal relation* for them or a *witness* to their global consistency. Honeyman, Ladner, and Yannakakis [15] showed that the universal relation problem is NP-complete, even for relations of arity 2.

On the structural side, the problem is to characterize when a collection of relations is globally consistent. It is easy to see that if the relations $R_1, \ldots, R_m$ are globally consistent, then they are pairwise consistent (i.e., every two of them are globally consistent). As pointed out in [15], however, the converse does not hold in general; in other words, pairwise consistency is a necessary but not sufficient condition for global consistency. This state of affairs raised the question: can we identify the settings in which pairwise consistency is both a necessary and sufficient condition for global consistency? Let $R_1, \ldots, R_m$ be a collection of relations over a schema with $X_1, \ldots, X_m$ as the sets of attributes. The sets $X_1, \ldots, X_m$ can be viewed as the hyperedges of a hypergraph. Beeri et al. [9] showed that the sets of attributes of a schema form an acyclic hypergraph if and only if the local-to-global consistency property for relations over that schema holds, which means that every collection of pairwise consistent relations over the schema is globally consistent. Thus, for acyclic schemas, pairwise consistency is necessary

and sufficient for global consistency. Consequently, the universal relation problem is solvable in polynomial time, if the sets of attributes of the schema form an acyclic hypergraph.

Much of the research in database theory assumes that relations are sets. In 1993, Chaudhuri and Vardi [12] pointed out that there is a gap between database theory and database practice because "real" databases use bags (multisets). They called for a re-examination of the foundations of databases where the fundamental concepts and algorithmic problems are investigated under bag semantics, instead of set semantics. In particular, Chaudhuri and Vardi [12] raised the question of the decidability of the conjunctive query containment problem under bags semantics (the same problem under set semantics is known to be NP-complete [11]). In spite of various efforts in the past and some recent progress [19, 20], this question remains unanswered at present.

It is perhaps surprising that a study of consistency notions under bag semantics has not been carried so far. Our main goal here is to embark on such a study and to explore both structural and algorithmic aspects of pairwise consistency and of global consistency under bag semantics. In this study, the consistency notions for bags are, of course, defined using bag semantics in the computation of projections.

Summary of Results In general, properties of relations need not carry over automatically to similar properties of bags. This phenomenon manifests itself in the context of consistency properties. Indeed, it is well known that if a collection of relations is globally consistent, then their relational join is a witness to their global consistency (see, e.g., [15]); in other words, their relational join is a universal relation for them and, in fact, it is the largest universal relation. In contrast, we point out that this property fails for bags, i.e., there is a collection of bags that is globally consistent but the bag-join of the bags in the collection is not a witness to their global consistency; furthermore, there may be no biggest witness to the consistency of these bags.

Our first result asserts that two bags are consistent if and only if they have the same projection on their common attributes. While the analogous fact for relations is rather trivial, here we need to bring in tools from the theory of linear programming and maximum flow problems. As a corollary, we obtain a polynomial-time algorithm for checking whether two given bags are consistent and returning a witness to their consistency, if they are consistent. After this, we establish our main result concerning the structure of bag consistency. Specifically, we show that the sets of attributes of a schema form an acyclic hypergraph if and only if the local-to-global consistency for bags over that schema holds. Thus, the main finding by Beeri et al. [9] about acyclicity and consistency extends to bags. The architecture of the proof, however, is different from that in [9]. In particular, if a schema is cyclic, we give an explicit construction of a collection of bags that are pairwise consistent, but not globally consistent; the inspiration for our construction comes from an earlier construction of hard-to-prove tautologies in propositional logic by Tseitin [23].

We then explore algorithmic aspects of global consistency for bags by analyzing the computational complexity of the global consistency problem for bags: given a collection of bags, are these bags globally consistent? Using a sparse-model property of integer programming that is reminiscent of Carathéodory's Theorem for conic hulls [13], we first show that this problem is in NP, even when the schema is part of the input. After this, we establish the following dichotomy theorem for fixed schemas: if the schema is acyclic, then the global consistency problem for bags is solvable in polynomial time, while if the schema is cyclic, then the global consistency problem for bags is NP-complete. The latter result contrasts sharply with the state of affairs for relations, where, for each fixed schema, the global consistency problem for relations is solvable in polynomial time. Our NP-hardness results build on an earlier NP-hardness result about three-dimensional statistical data tables by Irving and Jerrum [17], which was later on refined by De Loera and Onn [21]. Translated into our context, this result asserts the NP-hardness of the global consistency problem for bags over the triangle hypergraph, i.e., the hypergraph with hyperedges of the form $\{A_1, A_2\}, \{A_2, A_3\}, \{A_3, A_1\}$.

We conclude the paper with a brief overview of extensions of the results reported here to relations over semirings.

Related Work The interplay between local consistency and global consistency arises naturally in several different settings. Already in 1962, Vorob'ev [25] studied this interplay in the setting of probability distributions and characterized the local-to-global consistency property for probability distributions in terms of a structural property of hypergraphs that turned out to be equivalent to hypergraph acyclicity. It appears that Beeri et al. [9] were unaware of Vorob'ev work, but later on Vorob'ev's work was cited in a survey of database theory by Yannakakis [27]. In recent years, the interplay between local consistency and global consistency has been explored at great depth in the setting of quantum information by Abramsky and his collaborators (see, e.g., [3, 4, 5]). In that setting, the interest is in contextuality phenomena, which are situations where collections of measurements are locally consistent but globally inconsistent - Bell's celebrated theorem [10] is an instance of this. The similarities between these different settings (probability distributions, relational databases, and quantum mechanics) were pointed out explicitly by Abramsky [1, 2]. This also raised the question of developing a unifying framework in which, among other things, the results by Vorob'ev and the results by Beeri et al. are special cases of a single result. Using a relaxed notion of consistency, we established such a result for relations over semirings [6]. For the bag semiring, however, the relaxed notion of consistency that we studied in [6] is essentially equivalent to the consistency of probability distributions with rational values (and not to the consistency of bags). This left open the question of exploring the interplay between (the standard notions of) local consistency and global consistency for bags, which is what we set to do in the present paper.

2. RELATIONAL CONSISTENCY

Basic Notions An *attribute* A is a symbol with an associated set $\mathrm{Dom}(A)$ called its *domain*. If X is a finite set of attributes, then we write $\mathrm{Tup}(X)$ for the set of *X-tuples*; this means that $\mathrm{Tup}(X)$ is the set of functions that take each attribute $A \in X$ to an element of its domain $\mathrm{Dom}(A)$. Note that $\mathrm{Tup}(\emptyset)$ is non-empty as it contains the *empty tuple*, i.e., the unique function with empty domain. If $Y \subseteq X$ is a subset of attributes and t is an X-tuple, then the *projection of t on Y*, denoted by $t[Y]$, is the unique Y-tuple that agrees with t on Y. In particular, $t[\emptyset]$ is the empty tuple.

Let X be a set of attributes. A *relation* over X is a function $R : \mathrm{Tup}(X) \to \{0, 1\}$. We write $R(X)$ to emphasize

the fact that R is a relation over *schema* X. The *support* $\mathrm{Supp}(R)$ of R is the set of X-tuples t with a non-zero value, i.e., $\mathrm{Supp}(R) := \{t \in \mathrm{Tup}(X) : R(t) \neq 0\}$. Whenever no confusion arises, we write R' to denote $\mathrm{Supp}(R)$. We say that R is *finite* if its support R' is a finite set. In what follows, we will make the blanket assumption that all relations considered are finite, so we will omit the term "finite". Every relation R can be identified with its support R', thus every relation R can be viewed as a finite set of X-tuples.

Let R be a relation over X and assume that $Z \subseteq X$. The *projection* $R[Z]$ of R on Z is the relation over Z consisting of all projections $t[Z]$, for $t \in R$. It is easy to see that for all $W \subseteq Z \subseteq X$, we have $R[Z][W] = R[W]$).

If X and Y are sets of attributes, then we write XY as shorthand for the union $X \cup Y$. Accordingly, if x is an X-tuple and y is a Y-tuple such that that $x[X \cap Y] = y[X \cap Y]$, then we write xy to denote the XY-tuple that agrees with x on X and on y on Y. We say that x *joins with* y, and that y *joins with* x, to *produce* the tuple xy.

The *join* $R \bowtie S$ of two relations $R(X)$ and $S(Y)$ is the relation over XY consisting of all XY-tuples t such that $t[X]$ is in R and $t[Y]$ is in S, i.e., all tuples of the form xy such that $x \in R$, $y \in S$, and x joins with y.

Consistency of Two Relations Assume that $R(X)$ and $S(X)$ are two relations over the schemas X and Y. We say that $R(X)$ and $S(Y)$ are *consistent* if there is a relation T over XY such that $T[X] = R$ and $T[Y] = S$. We also say that T *witnesses* the consistency of R and S. The next proposition, whose proof is straightforward, gives a useful criterion for the consistency of R and S.

PROPOSITION 1. *Let $R(X)$ and $S(Y)$ be two relations. The following statements are equivalent:*

1. *R and S are consistent.*
2. *$R[X \cap Y] = S[X \cap Y]$.*

Global Consistency of Relations Let $R_1(X_1), \ldots, R_m(X_m)$ be relations over the schemas $X_1, \ldots, X_m$. We say that the collection $R_1, \ldots, R_m$ is *globally consistent* if there is a relation T over $X_1 \cup \cdots \cup X_m$ such that $R_i = T[X_i]$ for all $i \in [m] = \{1, \ldots, m\}$. We say that T *witnesses* the global consistency of $R_1, \ldots, R_m$, and we call it a *universal relation* for $R_1, \ldots, R_m$. The next result presents well known and easy to prove facts about global consistency (see, e.g., [15]).

PROPOSITION 2. *Assume that $R_1(X_1), \ldots, R_m(X_m)$ are relations over the schemas $X_1, \ldots, X_m$.*

- *If T is a relation witnessing the global consistency of the relations $R_1, \ldots, R_m$, then $T \subseteq R_1 \bowtie \cdots \bowtie R_m$.*
- *The collection $R_1, \ldots, R_m$ is globally consistent if and only if $(R_1 \bowtie \cdots \bowtie R_m)[X_i] = R_i$ for all $i = 1, \ldots, m$.*

Consequently, if the collection $R_1, \ldots, R_m$ is globally consistent, then the join $R_1 \bowtie \cdots \bowtie R_m$ is the *largest* universal relation for $R_1, \ldots, R_m$.

In relational database theory, there has been an extensive study of both the structural and the algorithmic aspects of global consistency. We begin by surveying some of the results concerning the structural aspects of global consistency. The main problem is to characterize when a collection of relations is globally consistent.

We say that the relations $R_1(X_1), \ldots, R_m(X_m)$ are *pairwise consistent* if for every $i, j \in [m]$, the relations $R_i(X_i)$ and $R_j(X_j)$ are consistent. Clearly, if a relation T witnesses the global consistency of $R_1, \ldots, R_m$, then the relation $T[X_iX_j]$ witnesses the consistency of R_i and R_j, for every $i, j \in [m]$. Thus, if the collection $R_1, \ldots, R_m$ is globally consistent, then the relations $R_1, \ldots, R_m$ are pairwise consistent. The converse, however, is not true, in general. Indeed, let $X_1 = \{A_1, A_2\}$, $X_2 = \{A_2, A_3\}$, $X_3 = \{A_3, A_1\}$ and consider the relations $R_1(A_1A_2) = \{00, 11\}$, $R_2(A_2A_3) = \{01, 10\}$, $R_3(A_3A_1) = \{00, 11\}$. By Proposition 1, these relations are pairwise consistent. By Proposition 2, however, they are not globally consistent because $R_1 \bowtie R_2 \bowtie R_3 = \emptyset$.

Beeri, Fagin, Maier, and Yannakakis [9] characterized the set of schemas for which pairwise consistency is a necessary and sufficient condition for global consistency of relations. Their characterization involves notions from hypergraph theory that we now review.

Acyclic Hypergraphs A *hypergraph* is a pair $H = (V, E)$, where V is a set of *vertices* and E is a set of *hyperedges*, each of which is a non-empty subset of V. Every collection $X_1, \ldots, X_m$ of sets of attributes can be identified with a hypergraph $H = (V, E)$, where $V = X_1 \cup \cdots \cup X_m$ and $E = \{X_1, \ldots, X_m\}$. Conversely, every hypergraph $H = (V, E)$ gives rise to a collection $X_1, \ldots, X_m$ of sets of attributes, where $X_1, \ldots, X_m$ are the hyperedges of H. Thus, we can move from collections of sets of attributes to hypergraphs, and vice versa. The notion of an *acyclic* hypergraph generalizes the notion of an acyclic graph. Since we will not work directly with the definition of an acyclic hypergraph, we refer the reader to [9] for the precise definition. Instead, we focus on other notions that are equivalent to hypergraph acyclicity and will be of interest to us in the sequel.

Conformal and Chordal Hypergraphs The *primal* graph of a hypergraph $H = (V, E)$ is the undirected graph that has V as its set of vertices and has an edge between any two distinct vertices that appear together in at least one hyperedge of H. A hypergraph H is *conformal* if the set of vertices of every clique (i.e., complete subgraph) of the primal graph of H is contained in some hyperedge of H. A hypergraph H is *chordal* if its primal graph is chordal, that is, if every cycle of length at least four of the primal graph of H has a chord (i.e., an edge that connects two nodes on the cycle, but is not one of the edges of the cycle). To illustrate these concepts, let $V_n = \{A_1, \ldots, A_n\}$ be a set of n vertices and consider the hypergraphs

$$P_n = (V_n, \{A_1, A_2\}, \ldots, \{A_{n-1}, A_n\}) \quad (1)$$

$$C_n = (V_n, \{A_1, A_2\}, \ldots, \{A_{n-1}, A_n\}, \{A_n, A_1\}) \quad (2)$$

$$H_n = (V_n, \{V_n \setminus \{A_i\} : 1 \leq i \leq n\}) \quad (3)$$

If $n \geq 2$, then the hypergraph P_n is both conformal and chordal. The hypergraph $C_3 = H_3$ is chordal, but not conformal. For every $n \geq 4$, the hypergraph C_n is conformal, but not chordal, while the hypergraph H_n is chordal, but not conformal.

Running Intersection Property A hypergraph H has the *running intersection property* if there is a listing $X_1, \ldots, X_m$ of all hyperedges of H such that for every $i \in [m]$ with $i \geq 2$, there exists a $j < i$ such that $X_i \cap (X_1 \cup \cdots \cup X_{i-1}) \subseteq X_j$.

Join Tree A *join tree* for a hypergraph H is an undirected tree T with the set E of the hyperedges of H as its vertices

and such that for every vertex v of H, the set of vertices of T containing v forms a subtree of T, i.e., if v belongs to two vertices X_i and X_j of T, then v belongs to every vertex of T in the unique simple path from X_i to X_j in T.

Local-to-Global Consistency Property for Relations Let H be a hypergraph and let $X_1, \ldots, X_m$ be a listing of all hyperedges of H. We say that H has the *local-to-global consistency property for relations* if every pairwise consistent collection $R_1(X_1), \ldots, R_m(X_m)$ of relations over the schemas $X_1, \ldots, X_m$ is globally consistent.

We are ready to state the main result in Beeri et al. [9].

THEOREM 1 (THEOREM 3.4 IN [9]). *Let H be a hypergraph. The following statements are equivalent:*

(a) H is an acyclic hypergraph.

(b) H is a conformal and chordal hypergraph.

(c) H has the running intersection property.

(d) H has a join tree.

(e) H has the local-to-global consistency property for relations.

As an illustration, if $n \geq 2$, the hypergraph P_n is acyclic, hence it has the local-to-global consistency property for relations. In contrast, if $n \geq 3$, the hypergraphs C_n and H_n are cyclic, hence they do not have the local-to-global consistency property for relations.

Complexity of Global Consistency for Relations We now discuss the algorithmic aspects of global consistency. The *global consistency problem for relations* (also known as the *universal relation problem for relations*) asks: given a hypergraph $H = (V, \{X_1, \ldots, X_m\})$ and relations $R_1, \ldots, R_m$ over H, is the collection $R_1, \ldots, R_m$ globally consistent? Honeyman, Ladner, and Yannakakis [15] established the following result.

THEOREM 2. *The global consistency problem for relations is* NP*-complete.*

The NP-hardness of the global consistency problem for relations is proved via a reduction from 3-COLORABILITY in which each relation has arity 2 and consists of just six pairs. Specifically, each edge (u, v) in a given graph G gives rise to a relation of arity 2 with attributes u and v; the six pairs in the relation are the pairs of different colors chosen from the three colors "red", "blue", and "green". The membership in NP uses the observation that if a collection $R_1, \ldots, R_m$ of relations is globally consistent, then a witness W of this fact can be obtained as follows: for each $i \leq m$ and each tuple $t \in R_i$, pick a tuple in the join $R_1 \bowtie \cdots \bowtie R_m$ that extends t and insert it in W. In particular, the cardinality $|W|$ of W is bounded by the sum $\sum_{i=1}^{m} |R_i| \leq m \max\{|R_i| : i \in [m]\}$, and thus the size of W is bounded by a polynomial in the size of the input hypergraph H and the input relations $R_1, \ldots, R_m$.

Several restricted cases of the global consistency problem for relations turn out to be solvable in polynomial time.

First, Proposition 1 implies that the consistency problem for two relations is solvable in polynomial time, since it amounts to checking that the two given relations $R(X)$ and $S(Y)$ have the same projection on $X \cap Y$.

Second, from the preceding fact and from Theorem 1, it follows that the global consistency problem for relations is solvable in polynomial time when restricted to acyclic hypergraphs, since, in this case, the global consistency of a collection of relations is equivalent to the pairwise consistency of the relations in the collection.

Finally, for every fixed hypergraph $H = (V, \{X_1, \ldots, X_m\})$ (be it cyclic or acyclic), the global consistency problem restricted to relations $R_1(X_1), \ldots, R_m(X_m)$ with sets of attributes $X_1, \ldots, X_m$ is also solvable in polynomial time. This is so because, by Proposition 2, one can first compute the join $J = R_1 \bowtie \cdots \bowtie R_m$ in polynomial time and then check whether $J[X_i] = R_i$ holds, for $i = 1, \ldots, m$. While the cardinality $|J|$ of this witness J can only be bounded by $\prod_{i=1}^{m} |R_i| \leq \max\{|R_i| : i \in [m]\}^m$, this cardinality is still polynomial in the size of the input because, in this case, the exponent m is fixed and not part of the input.

3. BAG CONSISTENCY

Basic Notions Let X be a set of attributes. A *bag* over X is a function $R : \mathrm{Tup}(X) \to \{0, 1, 2, \ldots\}$. As with relations, we write $R(X)$ to emphasize the fact that R is a bag over X; the support $\mathrm{Supp}(R)$ (also denoted by R') of R is the set of X-tuples t that are assigned non-zero value. We say that R is *finite* if its support R' is a finite set. In the sequel, we will assume that all bags are finite.

If R is a bag and t is an X-tuple, then the non-negative integer $R(t)$ is called the *multiplicity* of t in R; we write $t : R(t)$ to denote that the multiplicity of t in R is equal to $R(t)$. Every bag R can be viewed as a finite set of elements of the form $t : R(t)$, where $t \in R'$ and $R(t) \neq 0$. A bag can also be represented in tabular form. For example, the table

A	B	#
a_1	b_1 :	2
a_2	b_2 :	1
a_3	b_3 :	5

represents the bag $R = \{(a_1, b_1) : 2, (a_2, b_2) : 1, (a_3, b_3) : 5\}$. Let R be a bag over X and assume that $Z \subseteq X$. If t is a Z-tuple, then the *marginal of R over t* is defined by

$$R(t) := \sum_{\substack{r \in R' : \\ r[Z] = t}} R(r). \tag{4}$$

Thus, every bag R over X induces a bag over Z, called the *marginal of R on Z* and denoted by $R[Z]$. It is easy to verify that for all $W \subseteq Z \subseteq X$, we have $R[Z][W] = R[W]$.

Let R be a bag over X and S a bag over Y. The *bag join* $R \bowtie_b S$ of R and S is the bag over XY having support $R' \bowtie S'$ and such that every XY-tuple $t \in R' \bowtie S'$ has multiplicity $(R \bowtie_b S)(t) = R(t[X]) \times S(t[Y])$.

Consistency of Two Bags Two bags $R(X)$ and $S(Y)$ are *consistent* if there is a bag $T(XY)$ such that $T[X] = R$ and $T[Y] = S$, where now the projections are computed according to Equation (4); we say that T *witnesses* the consistency of R and S. It is easy to see that if $R(X)$ and $S(Y)$ are consistent bags and T is a bag that witnesses their consistency, then $T' \subseteq R' \bowtie S'$, that is, the support of T is contained in the join of the supports of R and S.

By Proposition 1, if two relations $R(X)$ and $S(Y)$ are consistent, then their join $R \bowtie S$ witnesses their consistency; moreover, $R \bowtie S$ is the largest relation that has this property. In contrast, this is not true for bags because there are consistent bags $R(X)$ and $S(Y)$ such that the support T' of every bag T witnessing their consistency is a proper subset of $R' \bowtie S'$. For example, consider the

bags $R_1(AB) = \{(1,2) : 1,\ (2,2) : 1\}$ and $S_1(BC) = \{(2,1) : 1,\ (2,2) : 1\}$; their consistency (as bags) is witnessed by the bags $T_1(ABC) = \{(1,2,2) : 1,\ (2,2,1) : 1\}$ and $T_2(ABC) = \{(1,2,1) : 1,\ (2,2,2) : 1\}$, but no other bag. This example can be extended as follows. For $n \geq 2$, let $R_{n-1}(A,B)$ and $S_{n-1}(B,C)$ be the bags

$$\{(1,2) : 1,\ (2,2) : 1,\ \ldots,\ (1,n) : 1, (n,n) : 1\}$$
$$\{(2,1) : 1,\ (2,2) : 1,\ \ldots,\ (n,1) : 1, (n,n) : 1\},$$

respectively. For every $n \geq 2$, the bags R_{n-1} and S_{n-1} are consistent. In fact, there are exactly 2^{n-1} bags witnessing their consistency; these witnesses are pairwise incomparable in the bag-containment sense and their supports are properly contained in the support $(R_{n-1} \bowtie_b S_{n-1})'$ of the bag join $R_{n-1} \bowtie_b S_{n-1}$. Note that the bags R_{n-1} and S_{n-1} are actually relations and that their join $R_{n-1} \bowtie S_{n-1}$ witnesses their consistency as relations, but not as bags.

By Proposition 1, two relations $R(X)$ and $S(Y)$ are consistent if and only if $R[X \cap Y] = S[X \cap Y]$. It is natural to ask if an analogous result holds true for bags. If two bags $R(X)$ and $S(Y)$ are consistent, then clearly $R[X \cap Y] = S[X \cap Y]$. The converse turns out to also be true, but its proof is far from obvious. We will establish the converse by bringing into the picture concepts from linear programming and from the theory of maximum flows.

With each pair of bags $R(X)$ and $S(Y)$, we associate the following linear program $P(R,S)$. Let $J = R' \bowtie S'$ be the join of the supports of R and S. For each $t \in J$, there is a variable x_t. For each $t \in J$ and $r \in R'$, define $a_{r,t} = 1$ if $t[X] = r$ and $a_{r,t} = 0$ if $t[X] \neq r$. Similarly, for each $t \in J$ and $s \in S'$, define $a_{s,t} = 1$ if $t[Y] = s$ and $a_{s,t} = 0$ if $t[Y] \neq s$. The constraints of $P(R,S)$ are:

$$\begin{array}{ll} \sum_{t \in J} a_{r,t} x_t = R(r) & \text{for } r \in R', \\ \sum_{t \in J} a_{s,t} x_t = S(s) & \text{for } s \in S', \\ x_t \geq 0 & \text{for } t \in J. \end{array} \tag{5}$$

The linear program $P(R,S)$ can be viewed as the set of the flow constraints of an instance of the max-flow problem. A *network* $N = (V, E, c, s, t)$ is a directed graph $G = (V, E)$ with a non-negative weight $c(u,v)$, called the *capacity*, assigned to each edge $(u,v) \in E$, and two distinguished vertices $s, t \in V$, called the *source* and the *sink*. A *flow* for the network is an assignment of non-negative weights $f(u,v)$ on the edges $(u,v) \in E$ so that both the capacity constraints and the flow constraints are respected, that is,

$$\begin{array}{ll} f(u,v) \leq c(u,v) & \text{for } (u,v) \in E, \\ \sum_{v \in N^-(u)} f(v,u) = \sum_{w \in N^+(u)} f(u,w) & \text{for } u \in V \setminus \{s,t\}, \end{array}$$

where $N^-(u)$ and $N^+(u)$ denote the sets of in-neighbors and out-neighbors of u in G. The *value* of such a flow is the quantity $\sum_{w \in N^+(s)} f(s,w) = \sum_{v \in N^-(t)} f(v,t)$, where the equality follows from the flow constraints. In the *max-flow problem*, the goal is to find a flow of maximum value. A flow is *saturated* if $f(s,w) = c(s,w)$ for every $w \in N^+(s)$ and $f(v,t) = c(v,t)$ for every $v \in N^-(t)$. It is obvious that if a saturated flow exists, then every max flow is saturated.

With each pair $R(X)$ and $S(Y)$ of bags, we associate the following network $N(R,S)$. The network has $1+|R'|+|S'|+1$ vertices: one source vertex s^*, one vertex for each tuple r in the support R' of R, one vertex for each tuple s in the support S' of S, and one target vertex t^*. There is an arc of capacity $R(r)$ from s^* to r for each $r \in R'$, an arc of capacity $S(s)$ from s to t^* for each $s \in S'$, and an arc of unbounded (i.e., very large) capacity from $t[X]$ to $t[Y]$ for each $t \in R' \bowtie S'$.

The next result yields several different characterizations of the consistency of two bags.

LEMMA 1. *Let $R(X)$ and $S(Y)$ be two bags. The following statements are equivalent:*

1. *$R(X)$ and $S(Y)$ are consistent.*
2. *$R[X \cap Y] = S[X \cap Y]$.*
3. *$P(R,S)$ is feasible over the rationals.*
4. *$P(R,S)$ is feasible over the integers.*
5. *$N(R,S)$ admits a saturated flow.*

PROOF. (*Sketch*) The equivalence of the statements (1) and (4) is immediate from the definitions. As discussed earlier, (1) implies (2). To show that (2) implies (3), we assume that $R[Z] = S[Z]$ and show that $P(R,S)$ is feasible over the rationals. For each $t \in J = R' \bowtie S'$, we set $x_t := R(t[X])S(t[Y])/R(t[Z]) = R(t[X])S(t[Y])/S(t[Z])$ (where the equality follows from the assumption that $R[Z] = S[Z]$) and verify that this is a rational solution of $P(R,S)$. For (3) implies (5), let $x^* = (x_t^*)_{t \in J}$ be a rational solution for $P(R,S)$ and let f be the following assignment for $N(R,S)$:

$$\begin{array}{ll} f(s^*, r) := c(s^*, r) = R(r) & \text{for each } r \in R'; \\ f(t[X], t[Y]) := x_t^* & \text{for each } t \in J; \\ f(s, t^*) := c(s, t^*) = S(s) & \text{for each } s \in S'. \end{array}$$

This assignment is a flow since the equations of $P(R,S)$ say that the flow-constraints are satisfied; furthermore, it is a saturated flow by construction. For (5) implies (1), let g be a saturated flow for $N(R,S)$; in particular, this is a max flow for $N(R,S)$. Since all capacities in $N(R,S)$ are integers, the integrality theorem for the max-flow problem asserts that there is a max flow f consisting of integers (see, e.g., [26]), which, of course, is also a saturated flow. Let $T(XY)$ be the bag defined by setting $T(t) := f(t[X], t[Y])$ for each $t \in R' \bowtie S'$. Since f is saturated, we have that $f(s^*, r) = c(s^*, r) = R(r)$ for each $r \in R'$ and $f(s, t^*) = c(s, t^*) = S(s)$ for each $s \in S'$. This means that the flow-constraints imply that T witnesses the consistency of R and S. Thus, the statements (1), (2), (3), and (5) are equivalent. □

The equivalence of statements (1) and (2) in Lemma 1 yields a simple polynomial-time test to determine the consistency of two bags, namely, given two bags $R(X)$ and $S(y)$, check whether or not $R[X \cap Y] = S[X \cap Y]$. Later on, we will see that the equivalence of statements (1) and (5) implies that there is a polynomial-time algorithm for constructing a witness to the consistency of two consistent bags.

Global Consistency for Bags Let $R_1(X_1), \ldots, R_m(X_m)$ be bags over the schemas $X_1, \ldots, X_m$. We say that the collection $R_1, \ldots, R_m$ is *globally consistent* if there a bag T over $X_1 \cup \cdots \cup X_m$ such that $T_i[X_i] = R_i$ for all $i \in [m]$. We say that the bag T *witnesses* the global consistency of the bags $R_1, \ldots, R_m$. As with relations, pairwise consistency of a collection of bags is a necessary, but not sufficient, condition for the global consistency of the collection. Let H be a hypergraph and let $X_1, \ldots, X_m$ be a listing of all hyperedges of H. We say that H has the

local-to-global consistency property for bags if every pairwise consistent collection $R_1(X_1), \ldots, R_m(X_m)$ of bags over the schemas $X_1, \ldots, X_m$ is globally consistent. The main structural result of this paper asserts that the acyclic hypergraphs are precisely the hypergraphs for which the local-to-global consistency property for bags holds.

THEOREM 3. *Let H be a hypergraph. The following statements are equivalent:*

(a) H is an acyclic hypergraph.

(b) H is a conformal and chordal hypergraph.

(c) H has the running intersection property.

(d) H has a join tree.

(e) H has the local-to-global consistency property for bags.

PROOF. (*Outline*) Let H be a hypergraph. By Theorem 1, statements (a), (b), (c), and (d) are equivalent, because these statements express "structural" properties of hypergraphs, i.e., they involve only the vertices and the hyperedges of the hypergraph at hand. So, we only have to show that statement (e), which involves "semantic" notions about bags, is equivalent to (one of) the other three statements. This will be achieved in two steps. First, we show that statement (c) implies statement (e), i.e., if H has the running intersection property, then H has the local-to-global consistency property for bags. Second, we show that statement (e) implies statement (b) by showing the contrapositive: if H is not conformal or H is not chordal, then H does not have the local-to-global consistency property for bags.

Step 1. If the hypergraph H has the running intersection property, then there is a listing $X_1, \ldots, X_m$ of its hyperedges such that for every $i \in [m]$ with $i \geq 2$, there is a $j \in [i-1]$ such that $X_i \cap (X_1 \cup \cdots \cup X_{i-1}) \subseteq X_j$. Let $R_1(X_1), \ldots, R_m(X_m)$ be a collection of pairwise consistent bags over the schemas $X_1, \ldots, X_m$. By induction on $i = 1, \ldots, m$, we show that there is a bag T_i over $X_1 \cup \cdots \cup X_i$ that witnesses the global consistency of the bags $R_1, \ldots, R_i$. The claim is obvious for the base case $i = 1$. Assume that $i \geq 2$ and that the claim is true for all smaller indices. Let $X := X_1 \cup \cdots \cup X_{i-1}$ and, by the running intersection property, let $j \in [i-1]$ be such that $X_i \cap X \subseteq X_j$. By induction hypothesis, there is a bag T_{i-1} over X that witnesses the global consistency of $R_1, \ldots, R_{i-1}$. We show that T_{i-1} and R_i are consistent by showing that $T_{i-1}[X \cap X_i] = R_i[X \cap X_i]$ and invoking Lemma 1. After this, we show that if T_i is a bag that witnesses the consistency of the bags T_{i-1} and R_i, then T_i witnesses the global consistency of $R_1, \ldots, R_i$.

Step 2. We have to show that if if H is not conformal or H is not chordal, then H does not have the local-to-global consistency property for bags. We first establish that it is enough to show that certain "minimal" hypergraphs do not have the local-to-global consistency property for bags. Specifically, it is enough to show the following two statements:

1. No hypergraph $H_n = (V_n, \{V_n \setminus \{A_i\} : 1 \leq i \leq n\})$ with $V_n = \{A_1, \ldots, A_n\}$ and $n \geq 3$ has the local-to-global consistency property for bags. Recall that H_n is not conformal.

2. No hypergraph $C_n = (V_n, \{\{A_i, A_{i+1}\} : i \in [n]\})$ with $V_n = \{A_1, \ldots, A_n\}$, $A_{n+1} := A_1$, and $n \geq 4$ has the local-to-global consistency property for bags. Recall that C_n is not chordal.

The preceding "minimal" non-conformal and non-chordal hypergraphs share the following properties: all their hyperedges have the same number of vertices and all their vertices appear in the same number of hyperedges. Let $H^* = (V^*, E^*)$ be a hypergraph and let d and k be positive integers. The hypergraph H^* is called *k-uniform* if every hyperedge of H^* has exactly k vertices. It is called *d-regular* if every vertex of H^* appears in exactly d hyperedges of H. Thus, the "minimal" non-conformal hypergraph H_n is $(n-1)$-uniform and $(n-1)$-regular. Likewise, the "minimal" non-chordal hypergraph C_n is 2-uniform and 2-regular.

Assume that H^* is a k-uniform and d-regular hypergraph with $d \geq 2$ and with hyperedges $E^* = \{X_1, \ldots, X_m\}$. We construct a collection $C(H^*) := \{R_1(X_1), \ldots, R_m(X_m)\}$ of bags and show that the bags in this collection are pairwise consistent but are not globally consistent. This will imply that the local-to-global consistency property for bags fails for the hypergraphs H_n and C_n above.

For each $i \in [m]$ with $i \neq m$, let R_i be the bag over X_i defined as follows: (a) the support R_i' of R_i consists of all tuples $t : X_i \to \{0, \ldots, d-1\}$ whose total sum $\sum_{C \in X_i} t(C)$ is congruent to 0 mod d; (b) $R_i(t) := 1$ for each such X_i-tuple, and $R_i(t) := 0$ for every other X_i-tuple.

For $i = m$, let R_m be the bag over X_m defined as follows: (a) the support R_m' of R_m consists of all tuples $t : X_m \to \{0, \ldots, d-1\}$ whose total sum $\sum_{C \in X_m} t(C)$ is congruent to 1 mod d; (b) $R_m(t) := 1$ for each such X_m-tuple, and $R_m(t) := 0$ for every other X_m-tuple.

To show that the bags $R_1, \ldots, R_m$ are pairwise consistent, it suffices (by Lemma 1) to show that for distinct $i, j \in [m]$, we have $R_i[Z] \equiv R_j[Z]$, where $Z := X_i \cap X_j$. In turn, this follows from the claim that for every $i \in [m]$ and every Z-tuple $t : Z \to \{0, \ldots, d-1\}$, we have $R_i(t) = d^{k-|Z|-1}$. Indeed, since by k-uniformity every hyperedge of H has exactly k vertices, for every $u \in \{0, \ldots, d-1\}$, there are exactly $d^{k-|Z|-1}$ many X_i-tuples $t_{i,u,1}, \ldots, t_{i,u,d^{k-|Z|-1}}$ that extend t and have total sum congruent to u mod d. It follows then that $R_i[Z] = R_j[Z]$ for every two distinct $i, j \in [m]$, regardless of whether $m \in \{i, j\}$ or $m \notin \{i, j\}$.

To show that the relations $R_1, \ldots, R_m$ are not globally consistent, we proceed by contradiction. If T were a bag that witnesses their consistency, then T would be non-empty and its support would contain a tuple t such that the projections $t[X_i]$ belong to the supports R_i' of the R_i, for each $i \in [m]$. In turn this means that

$$\sum_{C \in X_i} t(C) \equiv 0 \bmod d, \qquad \text{for } i \neq m \tag{6}$$

$$\sum_{C \in X_i} t(C) \equiv 1 \bmod d, \qquad \text{for } i = m. \tag{7}$$

Since by d-regularity each $C \in V$ belongs to exactly d many sets X_i, adding up all the equations in (6) and (7) gives

$$\sum_{C \in V} dt(C) \equiv 1 \bmod d, \tag{8}$$

which is absurd since the left-hand side is congruent to 0 mod d and the right-hand side is congruent to 1 mod d. □

It should be pointed out that the proof of Theorem 1 in [9] has a different architecture than the proof of our Theorem 3. In particular, to prove the equivalence between the local-to-global consistency property for relations and acyclicity, Beeri et al. make use of Graham's algorithm, which is an algorithm for testing if a given hypergraph is acyclic. More importantly, for every cyclic hypergraph H, the proof of

Theorem 1 in [9] yields a collection of relations over H that are pairwise consistent but not globally consistent; these relations, however, are not pairwise consistent as bags, therefore they cannot be used to prove Theorem 3.

As an immediate consequence of Theorems 1 and 3, we obtain the following result.

COROLLARY 1. *Let H be a hypergraph. The following statements are equivalent:*

(a) H has the local-to-global consistency property for relations.

(b) H has the local-to-global consistency property for bags.

Complexity of Global Consistency for Bags The *global consistency problem for bags* asks: given a hypergraph $H = (V, \{X_1, \ldots, X_m\})$ and bags $R_1, \ldots, R_m$ over H, is the collection $R_1, \ldots, R_m$ globally consistent? Using an integral version of Carathéodory's Theorem due to Eisenbrand and Shmonin [13], we can show that this problem is in NP.

At the end of Section 2, we saw that for every fixed hypergraph H, the global consistency problem for relations over the hyperedges of H is solvable in polynomial time. As we shall see next, the state of affairs is by far more nuanced for bags. Every fixed hypergraph H gives rise to the decision problem GCPB(H), which asks: given bags $R_1, \ldots, R_m$ over H, is the collection $R_1, \ldots, R_m$ globally consistent? The next result is a dichotomy theorem that classifies the complexity of all decision problems GCPB(H), where H is a hypergraph.

THEOREM 4. *Let $H = (V, \{X_1, \ldots, X_m\})$ be a hypergraph. Then the following statements are true.*

1. If H is acyclic, then GCPB(H) *is in* P.

2. If H is cyclic, then GCPB(H) *is* NP*-complete.*

PROOF. (*Hint*) The first part of the theorem follows from Lemma 1 and Theorem 3. For the second part of the theorem, NP-hardness is proved via a series of reductions.

We first show the NP-hardness of each of the problems GCPB(C_n) and GCPB(H_n), where $n \geq 3$, as follows.

The problem GCPB(C_3) generalizes the consistency problem for 3-dimensional contingency tables (3DCT): given a positive integer n and, for each $i, j, k \in [n]$, non-negative integers $R(i,k)$, $C(j,k)$, $F(i,j)$, is there an $n \times n \times n$ table of non-negative integers $X(i,j,k)$ such that $\sum_{q=1}^{n} X(i,q,k) = R(i,k)$, $\sum_{q=1}^{n} X(q,j,k) = C(j,k)$, $\sum_{q=1}^{n} X(i,j,q) = F(i,j)$ for all indices $i, j, k \in [n]$? This problem was shown to be NP-complete in [17]. To see that GCPB(C_3) generalizes the consistency problem for 3DCT, let X, Y, Z be three attributes with domain $[n]$, and let $R(XZ)$, $C(YZ)$, $F(XY)$ be the bags given by the three tables $R(i,k)$, $C(j,k)$, $F(i,j)$. Therefore, GCPB(C_3) is NP-hard. For $n \geq 4$, we show that there is a polynomial time reduction from GCPB(C_{n-1}) to GCPB(C_n). As for the problems GCPB(H_n), the problem GCPB(H_3) is NP-hard because $H_3 = C_3$; after this, we show that for every $n \geq 4$, there is a polynomial-time reduction from GCPB(H_{n-1}) to GCPB(H_n).

Finally, if H is a cyclic hypergraph, then we show that there exists some $n \geq 4$ such that GCPB(C_n) or GCPB(H_n) reduces in polynomial time to GCPB(H). □

Table 1 compares the structural and algorithmic aspects of global consistency for relations vs. those for bags.

4. RELATIONS OVER SEMIRINGS

What do relations and bags have in common? For quite some time, it has been realized that relations and bags can be viewed as different instances of a single generalized concept of a relation in which tuples have "labels" that come from the domain of some algebraic structure.

Ioannidis and Ramakrishnan [16] considered relations over *labeled systems* and studied the query containment problem for relations over such systems. Later on Green, Karvounarakis, and Tannen [14] considered relations over semirings and studied the provenance of query answers. A *semiring* is an algebraic structure of the form $K = (A, +, \times, 0, 1)$ such that $(A, +, 0)$ is a commutative monoid, $(A, \times, 1)$ is a monoid, $\times$ distributes over $+$, and $a \times 0 = 0 \times a = 0$, for every $a \in A$. A semiring K is *positive* if the following two properties hold: (i) if $a + b = 0$, then $a = 0$ and $b = 0$; (ii) if $a \times b = 0$, then $a = 0$ or $b = 0$ (i.e., K has no *zero divisors*). If K is a semiring and X is a set of attributes, then a *K-relation over X* is a function $R : \mathrm{Tup}(X) \to A$. Thus, relations are $\mathbb{B}$-relations where $\mathbb{B} = (\{0,1\}, \vee, \wedge, 0, 1)$ is the Boolean semiring, while bags are $\mathbb{N}$-relations, where $\mathbb{N} = (\{0, 1, 2, \ldots\}, +, \times, 0, 1)$ is the semiring of non-negative integers with the standard arithmetic operations.

In the PODS 2021 proceedings version of the present paper [7], we raised the question of whether or not the results about the global consistency for bags extend to K-relations, where K is a positive semiring. In particular, does the analog of Theorem 3 for K-relations hold, where K is an arbitrary positive semiring? If not, are there broad classes of semirings for which the analog of Theorem 3 for K-relations holds? Since that time, we have obtained fairly complete answers to these questions that we summarize next; these results will appear in a forthcoming paper.

Our first finding asserts that if K is an arbitrary positive semiring and H is a hypergraph such that the local-to-global consistency property for K-relations holds, then H must be acyclic. Thus, one of the two directions in Theorem 3 holds for arbitrary positive semirings. Our second finding, however, reveals that the reverse direction does not hold for arbitrary positive semirings. For this, we consider the semiring $\mathbb{R}_1 = (\{0\} \cup [1, \infty], +, \times, 0, 1)$ of real numbers that are either 0 or at least 1 and the acyclic hypergraph

$$H = (\{A, B, C, D), \{\{A, D\}, \{B, D\}, \{C, D\}\}).$$

We show that there are three $\mathbb{R}_1$-relations T_1, T_2, T_3 over H that are pairwise consistent but not globally consistent.

According to Proposition 1 and to Lemma 1, both relations and bags have the following property: two relations $R(X)$, $S(Y)$ (or two bags $R(X)$, $S(Y)$) are consistent if and only if $R[X \cap Y] = S[X \cap Y]$. We say that a semiring K has the *inner consistency* property if the preceding property holds for all pairs of K-relations. Our third finding tells that if K is a positive semiring with the inner consistency property and if H is an acyclic hypergraph, then the local-to-global consistency property holds for H. Thus, for positive semirings with the inner consistency property, the acyclicity of a hypergraph H is equivalent to the local-to-global consistency property for H. This result provides a common generalization of Theorem 1 for relations and of Theorem 3 for bags.

Finally, we identify several different sufficient conditions for a semiring to have the inner consistency property. As a result, we establish that the equivalence between acyclic-

Table 1: Relational Consistency vs. Bag Consistency

	Relations	Bags
Witness of global consistency	Join is a witnesses	Join need *not* be a witness
Local-to-global consistency property for H	H is acyclic	H is acyclic
Global Consistency Problem for acyclic H	in P	in P
Global Consistency Problem for cyclic H	in P	NP-complete

ity and the local-to-global consistency property holds for a plethora of semirings, including the tropical semirings, the log semirings, Łukasiewicsz' semiring, and every semiring that is a bounded distributive lattice.

Acknowledgments

The research of Albert Atserias was partially supported by MICIN project PID2019-109137GB-C22 (PROOFS). The research of Phokion Kolaitis was partially supported by NSF Award No. 1814152.

5. REFERENCES

[1] S. Abramsky. Relational databases and Bell's theorem. In *In Search of Elegance in the Theory and Practice of Computation - Essays Dedicated to Peter Buneman*, volume 8000 of *Lecture Notes in Computer Science*, pages 13–35. Springer, 2013.

[2] S. Abramsky. Contextual semantics: From quantum mechanics to logic, databases, constraints, and complexity. *Bull. EATCS*, 113, 2014.

[3] S. Abramsky, R. S. Barbosa, K. Kishida, R. Lal, and S. Mansfield. Contextuality, cohomology and paradox. In *24th EACSL Annual Conference on Computer Science Logic, CSL 2015*, pages 211–228, 2015.

[4] S. Abramsky and A. Brandenburger. A unified sheaf-theoretic account of non-locality and contextuality. *CoRR*, abs/1102.0264, 2011.

[5] S. Abramsky, S. Mansfield, and R. S. Barbosa. The cohomology of non-locality and contextuality. In *Proc. 8th Intern. Workshop on Quantum Physics and Logic, QPL 2011*, volume 95 of *EPTCS*, pages 1–14, 2011.

[6] A. Atserias and P. G. Kolaitis. Consistency, acyclicity, and positive semirings. *CoRR*, abs/2009.09488, 2020. To appear in a volume dedicated to Samson Abramsky in the series *Outstanding Contributions to Logic*.

[7] A. Atserias and P. G. Kolaitis. Structure and complexity of bag consistency. In *PODS'21: Proc. of the 40th ACM Symposium on Principles of Database Systems*, pages 247–259. ACM, 2021.

[8] C. Beeri, P. A. Bernstein, and N. Goodman. A sophisticate's introduction to database normalization theory. In *Fourth Intern. Conf. on Very Large Data Bases*, pages 113–124. IEEE Computer Society, 1978.

[9] C. Beeri, R. Fagin, D. Maier, and M. Yannakakis. On the desirability of acyclic database schemes. *J. ACM*, 30(3):479–513, July 1983.

[10] J. S. Bell. On the Einstein-Podolsky-Rosen paradox. *Physics Physique Fizika*, 1(3):195, 1964.

[11] A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *Proc. of the 9th Annual ACM Symp. on Theory of Computing*, pages 77–90, 1977.

[12] S. Chaudhuri and M. Y. Vardi. Optimization of *Real* conjunctive queries. In *Proc. of the Twelfth ACM Symposium on Principles of Database Systems*, pages 59–70, 1993.

[13] F. Eisenbrand and G. Shmonin. Carathéodory bounds for integer cones. *Operations Research Letters*, 34(5):564 – 568, 2006.

[14] T. J. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *Proc. of the Twenty-Sixth ACM Symposium on Principles of Database Systems*, pages 31–40, 2007.

[15] P. Honeyman, R. E. Ladner, and M. Yannakakis. Testing the universal instance assumption. *Inf. Process. Lett.*, 10(1):14–19, 1980.

[16] Y. E. Ioannidis and R. Ramakrishnan. Containment of conjunctive queries: Beyond relations as sets. *ACM Trans. Database Syst.*, 20(3):288–324, 1995.

[17] R. W. Irving and M. Jerrum. Three-dimensional statistical data security problems. *SIAM J. Comput.*, 23(1):170–184, 1994.

[18] W. Kent. Consequences of assuming a universal relation. *ACM Trans. Database Syst.*, 6(4):539–556, 1981.

[19] M. A. Khamis, P. G. Kolaitis, H. Q. Ngo, and D. Suciu. Bag query containment and information theory. In *Proc. of the 39th ACM Symposium on Principles of Database Systems, PODS 2020*, pages 95–112, 2020.

[20] G. Konstantinidis and F. Mogavero. Attacking Diophantus: Solving a special case of bag containment. In *Proc. of the 38th ACM Symposium on Principles of Database Systems, PODS*, pages 399–413, 2019.

[21] J. A. D. Loera and S. Onn. The complexity of three-way statistical tables. *SIAM J. Comput.*, 33(4):819–836, 2004.

[22] D. Maier, J. D. Ullman, and M. Y. Vardi. On the foundations of the universal relation model. *ACM Trans. Database Syst.*, 9(2):283–308, 1984.

[23] G. S. Tseitin. On the complexity of derivation in propositional calculus. *Structures in Constructive Mathematics and Mathematical Logic*, pages 115–125, 1968.

[24] J. D. Ullman. The U.R. strikes back. In *Proc. of the ACM Symposium on Principles of Database Systems*, pages 10–22, 1982.

[25] N. N. Vorob'ev. Consistent families of measures and their extensions. *Theory of Probability & Its Applications*, 7(2):147–163, 1962.

[26] H. S. Wilf. *Algorithms and complexity (2. ed.)*. A K Peters, 2002.

[27] M. Yannakakis. Perspectives on database theory. *SIGACT News*, 27(3):25–49, 1996.

Technical Perspective: Model Counting Meets Distinct Elements in a Data Stream

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

Model counting is the problem of approximately counting the number $|\mathrm{Sol}(\phi)|$ of satisfying assignments to a given model ϕ, which could, for example, be a formula in conjunctive normal form (CNF) or a formula in disjunctive normal form (DNF). One is interested in an (ϵ, δ)-approximation scheme, which is a randomized algorithm which outputs a number X for which $(1-\epsilon)|\mathrm{Sol}(\phi)| \leq X \leq (1+\epsilon)|\mathrm{Sol}(\phi)|$ with probability at least $1-\delta$. There are fully polynomial-time (ϵ, δ)-approximation schemes for DNFs, while for CNFs it is NP-hard to devise such a scheme. Nevertheless, such a scheme is possible for CNFs with a small number $(\mathrm{poly}(|\phi|, 1/\epsilon, 1/\delta))$ of calls to an NP-oracle. This often suffices in practice using efficient SAT solvers to replace the oracle calls.

Seemingly unrelated to model counting is the problem of computing over data streams. Here an algorithm is given a stream $a_1, a_2, \ldots, a_m$ of m elements, each drawn from a universe $\{0,1\}^n$ of $N = 2^n$ items, and the goal is to estimate statistics of the stream using low memory with a single pass over the data stream. A statistic of particular importance is the number F_0 of distinct elements. Computing F_0 exactly requires $\Omega(\min(m, 2^n))$ memory. However, there are very low memory (ϵ, δ)-approximation schemes for F_0.

The notion of efficiency for model counting is very different than that for estimating the number of distinct elements in a data stream. In model counting one wants to minimize the *time complexity*, or if the problem is NP-hard, one seeks to minimize the number of NP-oracle calls. In contrast, in the data stream model often the main goal is to minimize the *space complexity*, i.e., the memory required, to estimate F_0. It is not at all clear that these two models are related.

The paper "Model Counting Meets F_0 Estimation" [4] by A. Pavan, N. V. Vinodchandran, A. Bhattacharyya, and K.S. Meel is a beautiful work which gives a generic transformation of data stream algorithms for F_0-estimation to algorithms for model counting. The authors also show a converse to their generic transformation. Namely, by, framing F_0-estimation as a special case of DNF model counting, the authors obtain a generic algorithm for tasks such as F_0-estimation over affine spaces and DNF sets.

The starting point of the above paper is the observation that a hashing-based technique of Stockmeyer for model counting for CNFs [5], extended by Chakraborty, Meel, and Vardi to DNFs [2], actually uses the same techniques as an F_0-estimation data stream algorithm of Gibbons and Tirthapura [3].

The idea for model counting for CNFs is to hash all possible solutions (assignments to variables) using a hash function h to the range $\{0,1\}^m$. Invoking an NP oracle multiple times, one can find the smallest value of m for which $h^{-1}(0^m)$ contains a number t between $c\epsilon^{-2}$ solutions and $2c\epsilon^{-2}$ solutions, for a constant $c > 0$, at which point one can estimate the total number of solutions as $t \cdot 2^m$, which can be shown to give an (ϵ, δ)-approximation scheme.

The idea for F_0-estimation in a data stream is to use a hash function h to map the universe $\{0,1\}^n$ to $\{0,1\}^n$ but to first restrict the range to $\{0,1\}^m$ for an $m \leq n$. Then one stores the set $h^{-1}(0^m)$ of preimages, and if this set size exceeds $2c\epsilon^{-2}$, one increases m by 1 and only retains those items in the current set that are also in the set $h^{-1}(0^{m+1})$ of preimages. In this way, one eventually finds an m for which the set $h^{-1}(0^m)$ of preimages contains a number t between $c\epsilon^{-2}$ distinct items and $2c\epsilon^{-2}$ distinct items, and can estimate F_0 as $t \cdot 2^m$, which like the algorithm for model counting, can be shown to given an (ϵ, δ)-approximation scheme.

Inspired by this striking similarity, Pavan et al. [4] give a generic transformation which captures all three distinct element algorithms in [1]. One can show that implementing the various distinct element algorithms efficiently in the context of model counting boils down to finding the minimum value that a hash function takes over a structured set of values, such as the solutions to a CNF or DNF formula. This can be done efficiently by choosing certain linear algebraic hash functions (affine maps with a Toeplitz structure for efficiency) and performing Gaussian elimination.

1. REFERENCES

[1] Ziv Bar-Yossef, T. S. Jayram, Ravi Kumar, D. Sivakumar, and Luca Trevisan. Counting distinct elements in a data stream. In RANDOM, 2002.
[2] Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. Algorithmic improvements in approximate counting for probabilistic inference: From linear to logarithmic SAT calls. In IJCAI/AAAI, 2016.
[3] Phillip B. Gibbons and Srikanta Tirthapura. Estimating simple functions on the union of data streams. In SPAA, 2001.
[4] Aduri Pavan, N. V. Vinodchandran, Arnab Bhattacharya, and Kuldeep S. Meel. Model counting meets f_0 estimation. In PODS, 2021.
[5] Larry J. Stockmeyer. The complexity of approximate counting. In STOC, 1983.

Model Counting Meets Distinct Elements in a Data Stream

[Extended Abstract]

A. Pavan
Iowa State University

N. V. Vinodchandran
University of Nebraska, Lincoln

Arnab Bhattacharyya
National University of Singapore

Kuldeep S. Meel
National University of Singapore

ABSTRACT

Constraint satisfaction problems (CSPs) and data stream models are two powerful abstractions to capture a wide variety of problems arising in different domains of computer science. Developments in the two communities have mostly occurred independently and with little interaction between them. In this work, we seek to investigate whether bridging the seeming communication gap between the two communities may pave the way to richer fundamental insights. To this end, we focus on two foundational problems: model counting for CSPs and computation of zeroth frequency moments (F_0) for data streams.

Our investigations lead us to observe striking similarity in the core techniques employed in the algorithmic frameworks that have evolved separately for model counting and F_0 computation. We design a recipe for translation of algorithms developed for F_0 estimation to that of model counting, resulting in new algorithms for model counting. We then observe that algorithms in the context of distributed streaming can be transformed to distributed algorithms for model counting. We next turn our attention to viewing streaming from the lens of counting and show that framing F_0 estimation as a special case of #DNF counting allows us to obtain a general recipe for a rich class of streaming problems, which had been subjected to case-specific analysis in prior works.

1. INTRODUCTION

Constraint Satisfaction Problems (CSP's) and the *data stream model* are two core themes in computer science with a diverse set of applications in topics including probabilistic reasoning, networks, databases, and verification. *Model counting* and computation of *zeroth frequency moment* (F_0) are fundamental problems for CSPs and the data stream model respectively. This paper is motivated by our observation that despite the usage of similar algorithmic techniques for the two problems, the developments in the two communities have, surprisingly, evolved separately, and rarely has a paper from one community been cited by the other.

Given a set of constraints φ over a set of variables in a finite domain $\mathcal{D}$, the problem of model counting is to estimate the number of solutions of φ. We are often interested when φ is restricted to a special class of representations such as Conjunctive Normal Form (CNF) and Disjunctive Normal Form (DNF). A data stream over a domain $[N]$ is represented by $\mathbf{a} = \langle a_1, a_2, \cdots, a_m \rangle$ where each item a_i is a subset of $[N]$. The *zeroth frequency moment*, denoted as F_0, of $\mathbf{a}$ is the number of distinct domain elements appearing in $\mathbf{a}$, i.e., $|\cup_i a_i|$ (traditionally, a_is are singletons; we will also be interested in the case when a_is are sets). The fundamental nature of model counting and F_0 computation over data streams has led to intense interest from theoreticians and practitioners alike in the respective communities for the past few decades.

The starting point of this work is the confluence of two viewpoints. The first viewpoint contends that some of the algorithms for model counting can conceptually be thought of as operating on the stream of the solutions of the constraints. The second viewpoint contends that a stream can be viewed as a DNF formula, and the problem of F_0 estimation is similar to model counting. These viewpoints make it natural to believe that algorithms developed in the streaming setting can be directly applied to model counting, and vice versa. We explore this connection and indeed design new algorithms for model counting inspired by algorithms for estimating F_0 in data streams. By exploring this connection further, we design new algorithms to estimate F_0 for streaming sets that are succinctly represented by constraints. It is worth noting that the two communities focus on seemingly different efficiency objectives: in streaming algorithms, space complexity is of major concern while in the context of model counting, time (especially NP query complexity in the context of CNF formulas) is of primary concern. Therefore, it is striking to observe that our trans-

This is a minor revision of the paper entitled "Model Counting Meets F_0 Estimation", Proceedings of the 40th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, Virtual Event, China, June 20-25, 2021. ACM 2021, ISBN 978-1-4503-8381-3. https://dl.acm.org/doi/10.1145/3452021.3458311

formation recipe leads to the design of *efficient* algorithms for F_0 estimation as well as model counting wherein *efficient* is measured by the concern of the corresponding community. We further investigate this observation and demonstrate that the space complexity of streaming algorithms provides an upper bound on the query complexity of model counting algorithms.

To put our contributions in context, we briefly survey the historical development of algorithmic frameworks in both model counting and F_0 estimation and point out the similarities.

Model Counting

The complexity-theoretic study of model counting was initiated by Valiant who showed that this problem, in general, is #P-complete [32]. This motivated researchers to investigate approximate model counting and in particular to design (ε, δ)-approximation schemes. The complexity of approximate model counting depends on its representation. When the model φ is represented as a CNF formula φ, designing an efficient (ε, δ)-approximation is NP-hard [30]. In contrast, when it is represented as a DNF formula, model counting admits an FPRAS (fully polynomial-time approximation scheme) [21, 22]. We will use #CNF to refer to the case when φ is a CNF formula and #DNF to refer to the case when φ is a DNF formula.

For #CNF, Stockmeyer [30] provided a hashing-based randomized procedure that can compute an (ε, δ)-approximation with running time $\text{poly}(|\varphi|, 1/\varepsilon, 1/\delta)$, given access to an NP oracle. Building on Stockmeyer's approach and motivated by the unprecedented breakthroughs in the design of SAT solvers, researchers have proposed a series of algorithmic improvements that have allowed the hashing-based techniques for approximate model counting to scale to formulas involving hundreds of thousands of variables. The practical implementations substitute the NP oracle with a SAT solver. In the context of model counting, we are primarily interested in time complexity and therefore, the number of NP queries is of key importance. The emphasis on the number of NP calls also stems from practice as the practical implementation of model counting algorithms have shown to spend over 99% of their time in the underlying SAT calls [29].

Karp and Luby [21] proposed the first FPRAS scheme for #DNF, which was improved in subsequent works [22, 9]. Chakraborty, Meel, and Vardi [5] demonstrated that the hashing-based framework can be extended to #DNF, thereby providing a unified framework for both #CNF and #DNF. Meel, Shrotri, and Vardi [24, 25] subsequently improved the complexity of the hashing-based approach for #DNF and observed that hashing-based techniques achieve better scalability than Monte Carlo techniques.

Zeroth Frequency Moment Estimation

Estimating (ε, δ)-approximation of the k^{th} frequency moments (F_k) of a stream is a central problem in the data streaming model [1]. In particular, considerable work has been done in designing algorithms for estimating the 0^{th} frequency moment (F_0), the number of distinct elements in the stream. For streaming algorithms, the primary resource concerns are space complexity and processing time per element. In general, for a streaming algorithm to be considered efficient, these should be $\text{poly}(\log N, 1/\epsilon)$ where N is the size of the universe (we assume δ to be a small constant and ignore $O(\log(\frac{1}{\delta}))$ factors in this discussion).

The first algorithm for computing F_0 with a constant factor approximation was proposed by Flajolet and Martin, who assumed the existence of hash functions with ideal properties resulting in an algorithm with undesirable space complexity [15]. In their seminal work, Alon, Matias, and Szegedy designed an $O(\log N)$ space algorithm for F_0 with a constant approximation ratio that employs 2-universal hash functions [1]. Subsequent investigations into hashing-based schemes by Gibbons and Tirthapura [16] and Bar-Yossef, Kumar, and Sivakumar [3] provided (ε, δ)-approximation algorithms with space and time complexity $\log N \cdot \text{poly}(\frac{1}{\varepsilon})$. Later, Bar-Yossef et al. proposed *three algorithms* with improved space and time complexity [2]. While the three algorithms employ hash functions, they differ conceptually in the usage of relevant random variables for the estimation of F_0. This line of work resulted in the development of an algorithm with optimal space complexity $O(\log N + \frac{1}{\varepsilon^2})$ and $O(\log N)$ update time to estimate the F_0 of a stream [20].

The above-mentioned works are in the setting where each data item a_i is an element of the universe. Subsequently, there has been a series of results of estimating F_0 in rich scenarios with a particular focus to handle the cases $a_i \subseteq \{1, 2, \cdots, N\}$ such as a list or a multidimensional range [3, 26, 31].

The Road to a Unifying Framework

As mentioned above, the algorithmic developments for model counting and F_0 estimation have largely relied on the usage of hashing-based techniques and yet these developments have, surprisingly, been separate, and rarely has a work from one community been cited by the other. In this context, we wonder whether it is possible to bridge this gap and if such an exercise would contribute to new algorithms for model counting as well as for F_0 estimation? The main conceptual contribution of this work is an affirmative answer to the above question. First, we point out that the two well-known algorithms; Stockmeyer's #CNF algorithm [30] that is further refined by Chakraborty et al. [5] and Gibbons and Tirthapura's F_0 estimation algorithm [16], are essentially the same.

The core idea of the hashing-based technique of Stockmeyer's and Chakraborty et al's scheme is to use pairwise independent hash functions to partition the solution space (satisfying assignments of a CNF formula) into *roughly equal and small* cells, wherein a cell is *small* if the number of solutions is less than a pre-computed threshold, denoted by Thresh. Then a good estimate for the number of solutions is the *number of solutions in an arbitrary cell* $\times$ *number of cells*. To determine the appropriate number of cells, the solution space is iteratively partitioned as follows. At the m^{th} iteration, a hash function with range $\{0,1\}^m$ is considered resulting in cells $h^{-1}(y)$ for each $y \in \{0,1\}^m$. An NP oracle can be employed to check whether a particular cell (for example $h^{-1}(0^m)$) is small by enumerating solutions one by one until we have either obtained Thresh+1 number of solutions or we have exhaustively enumerated all the solutions. If the cell $h^{-1}(0^m)$ is small, then the algorithm outputs $t \times 2^m$ as an estimate where t is the number of solutions in the cell $h^{-1}(0^m)$. If the cell $h^{-1}(0^m)$ is not small, then the algorithm moves on to the next iteration where a hash function with range $\{0,1\}^{m+1}$ is considered.

We now describe Gibbons and Tirthapura's algorithm for

F_0 estimation which we call the Bucketing algorithm. Without loss of generality, we assume that N is a power of two thus identify $[N]$ with $\{0,1\}^n$. The algorithm maintains a bucket of size Thresh and starts by picking a hash function $h : \{0,1\}^n \to \{0,1\}^n$. It iterates over sampling levels. At level m, when a data item x comes, if $h(x)$ starts with 0^m, then x is added to the bucket. If the bucket overflows, then the sampling level is increased to $m+1$ and all elements x in the bucket other than the ones with $h(x) = 0^{m+1}$ are deleted. At the end of the stream, the value $t \times 2^m$ is output as the estimate where t is the number of elements in the bucket and m is the sampling level.

These two algorithms are conceptually the same. In the Bucketing algorithm, at the sampling level m, it looks at only the first m bits of the hashed value; this is equivalent to considering a hash function with range $\{0,1\}^m$. Thus the bucket is nothing but all the elements in the stream that belong to the cell $h^{-1}(0^m)$. The final estimate is the number of elements in the bucket times the number of cells, identical to Chakraborty et al.'s algorithm. In both algorithms, to obtain an (ε, δ) approximation, the Thresh value is chosen as $O(\frac{1}{\varepsilon^2})$. To reduce the error probability to $1/\delta$,the median of $O(\log \frac{1}{\delta})$ independent estimations is output.

Our Contributions

Motivated by the conceptual identity between the two algorithms, we further explore the connections between algorithms for model counting and F_0 estimation.

First, we formalize a recipe to transform streaming algorithms for F_0 estimation to those for model counting. Such a transformation yields new (ε, δ)-approximate algorithms for model counting, which are different from currently known algorithms. Our transformation recipe from F_0 estimation to model counting allows us to view the problem of the design of distributed #DNF algorithms through the lens of *distributed functional monitoring* that is well studied in the data streaming literature.

Building on the connection between model counting and F_0 estimation algorithms, we design new algorithms to estimate F_0 over *structured set streams* where each element of the stream is a (succinct representation of a) subset of the universe. Thus, the stream is $S_1, S_2, \cdots$ where each $S_i \subseteq [N]$ and the goal is to estimate the F_0 of the stream, i.e. the size of $\cup_i S_i$. In this scenario, the goal is to design algorithms whose per-item time (time to process each S_i) is poly-logarithmic in the size of the universe. Structured set streams that are considered in the literature include 1-dimensional and multidimensional ranges [26, 31]. Several interesting problems, including max-dominance norm [6] and counting triangles in graphs [3], can be reduced to computing F_0 over such ranges.

We observe that several structured sets can be represented as small DNF formulae and thus F_0 counting over these structured set data streams can be viewed as a special case of #DNF. Using the hashing-based techniques for #DNF, we obtain a general recipe for a rich class of structured sets that include DNF sets, affine spaces, and multidimensional ranges. Prior work on structured sets had to rely on involved analysis for each of the specific instances, while our work provides a general recipe for both analysis and implementation.

A natural question that arises from the transformation is the relationship between the space complexity of the streaming algorithms and the query complexity of the obtained model counting algorithms. We establish a relationship between these two quantities by showing that the space complexity is an upper bound on the query complexity.

2. NOTATION

We will assume the universe $[N] = \{0,1\}^n$. *F_0 Estimation:* A data stream $\mathbf{a}$ over domain $[N]$ can be represented as $\mathbf{a} = a_1, a_2, \ldots, a_m$ wherein each item $a_i \in [N]$. Let $\mathbf{a}_u = \cup_i \{a_i\}$. F_0 of the stream $\mathbf{a}$ is $|\mathbf{a}_u|$. We are interested in a *probably approximately correct* scheme that returns an (ε, δ)-*estimate* c of F_0, i.e.,

$$\Pr\left[\frac{|\mathbf{a}_u|}{1+\varepsilon} \leq c \leq (1+\varepsilon)|\mathbf{a}_u|\right] \geq 1-\delta.$$

Model Couting: Let $X = \{x_1, x_2, \ldots, x_n\}$ be a set of Boolean variables. For a Boolean formula φ over variables X, let $\mathsf{Sol}(\varphi)$ denote the set of all satisfying assignments of φ. The *propositional model counting problem* is to compute $|\mathsf{Sol}(\varphi)|$ for a given formula φ. As in the case of F_0, we are interested in a *probably approximately correct* algorithm that takes as inputs a formula φ, a tolerance $\varepsilon > 0$, and a confidence $\delta \in (0,1]$, and returns a (ε, δ)-estimate c of $|\mathsf{Sol}(\varphi)|$ i.e., $\Pr\left[\frac{|\mathsf{Sol}(\varphi)|}{1+\varepsilon} \leq c \leq (1+\varepsilon)|\mathsf{Sol}(\varphi)|\right] \geq 1-\delta$.

k-wise Independent hash functions: Let $n, m \in \mathbb{N}$ and $\mathcal{H}(n,m) \triangleq \{h : \{0,1\}^n \to \{0,1\}^m\}$ be a family of hash functions mapping $\{0,1\}^n$ to $\{0,1\}^m$.

Definition 1. *A family of hash functions $\mathcal{H}(n,m)$ is k-wise independent, denoted $\mathcal{H}_{\mathsf{k\text{-}wise}}(n,m)$, if $\forall \alpha_1, \alpha_2, \ldots, \alpha_k \in \{0,1\}^m$, for all distinct $x_1, x_2, \ldots x_k \in \{0,1\}^n$,*

$$\Pr_{h \in \mathcal{H}(n,m)}[(h(x_1) = \alpha_1) \wedge (h(x_2) = \alpha_2) \ldots (h(x_k) = \alpha_k)] = \frac{1}{2^{km}}$$

Explicit families. An explicit hash family that we use is $\mathcal{H}_{\mathsf{Toeplitz}}(n,m)$, which is known to be 2-wise independent [4]. The family is defined as follows: $\mathcal{H}_{\mathsf{Toeplitz}}(n,m) \triangleq \{h : \{0,1\}^n \to \{0,1\}^m\}$ is the family of functions of the form $h(x) = Ax + b$ with $A \in \mathbb{F}_2^{m \times n}$ and $b \in \mathbb{F}_2^{m \times 1}$, and $\mathbb{F}_2$ is the finite field of size 2. For a hash function $h : \{0,1\}^n \to \{0,1\}^m$, $h_\ell : \{0,1\}^n \to \{0,1\}^\ell$, $\ell \in \{1, \ldots, m\}$, is the function where $h_\ell(y)$ is the first ℓ bits of $h(y)$.

3. FROM STREAMING TO COUNTING

As a first step, we present a unified view of the three hashing-based algorithms proposed in Bar-Yossef et al [2]. Their first algorithm, the Bucketing algorithm discussed above, is a modification of an F_0 estimation algorithm due to Gibbons and Tirthapura [16]. The second algorithm, which we call Minimum, is based on the idea that if we hash all the items of the stream, then $\mathcal{O}(1/\varepsilon^2)$-th minimum of the hash values can be used to compute a good estimate of F_0. The third algorithm, which we call Estimation, chooses a set of k functions, $\{h_1, h_2, \ldots, h_k\}$, such that each h_j is picked randomly from an $\mathcal{O}(\log(1/\varepsilon))$-independent hash family. For each hash function h_j, we say that h_j is not *lonely* if there exists $a_i \in \mathbf{a}$ such that $h_j(a_i) = 0$. One can then estimate F_0 of $\mathbf{a}$ by estimating the number of hash functions that are not lonely.

Algorithm 1, called ComputeF0, presents the overarching architecture of the three proposed algorithms. The architecture of ComputeF0 is fairly simple: it chooses a collection of

hash functions using ChooseHashFunctions, calls the subroutine ProcessUpdate for every incoming element of the stream and invokes ComputeEst at the end of the stream to return the F_0 approximation.

ChooseHashFunctions. As shown in Algorithm 2, the hash functions depend on the strategy being implemented. The subroutine PickHashFunctions$(\mathcal{H}, t)$ returns a collection of t independently chosen hash functions from the family $\mathcal{H}$. We use H to denote the collection of hash functions returned, this collection is viewed as either a 1-dimensional array or as a 2-dimensional array. When H is a 1-dimensional array, $H[i]$ denotes the ith hash function of the collection and when H is a 2-dimensional array $H[i][j]$ is the $[i,j]$th hash function.

ProcessUpdate. For a new item x, the update of $\mathcal{S}$, as shown in Algorithm 3 is as follows:

Bucketing For a new item x, if $H[i]_{m_i}(x) = 0^{m_i}$, then we add it to $\mathcal{S}[i]$ if x is not already present in $\mathcal{S}[i]$. If the size of $\mathcal{S}[i]$ is greater than Thresh (which is set to be $\mathcal{O}(1/\varepsilon^2)$), then we increment the m_i as in line 8.

Minimum For a new item x, if $H[i](x)$ is smaller than $\max \mathcal{S}[i]$, then we replace $\max \mathcal{S}[i]$ with $H[i](x)$.

Estimation For a new item x, compute $z = \mathsf{TrailZero}(H[i,j](x))$, i.e, the number of trailing zeros in $H[i,j](x)$, and replace $\mathcal{S}[i,j]$ with z if z is larger than $\mathcal{S}[i,j]$.

ComputeEst. Finally, for each of the algorithms, we estimate F_0 based on the sketch $\mathcal{S}$ as described in the subroutine ComputeEst presented as Algorithm 4. It is crucial to note that the estimation of F_0 is performed solely using the sketch $\mathcal{S}$ for the Bucketing and Minimum algorithms. The Estimation algorithm requires an additional parameter r that depends on a loose estimate of F_0.

Algorithm 1 ComputeF0(n, ε, δ)

```
1: Thresh ← 96/ε²
2: t ← 35 log(1/δ)
3: H ← ChooseHashFunctions(n, Thresh, t)
4: S ← {}
5: while true do
6:     if EndStream then exit;
7:     x ← input()
8:     ProcessUpdate(S, H, x, Thresh)
9: Est ← ComputeEst(S, Thresh)
10: return Est
```

Sketch Properties. For each of the three algorithms, their corresponding sketches can be viewed as arrays of size $35\log(1/\delta)$. The parameter Thresh is set to $96/\varepsilon^2$.

Bucketing The element $\mathcal{S}[i]$ is a tuple $\langle \ell_i, m_i \rangle$ where ℓ_i is a list of size at most Thresh, where $\ell_i = \{x \in \mathbf{a} \mid H[i]_{m_i}(x) = 0^{m_i}\}$. We use $\mathcal{S}[i](0)$ to denote ℓ_i and $\mathcal{S}[i](1)$ to denote m_i.

Minimum Each $\mathcal{S}[i]$ holds the lexicographically Thresh many smallest elements of $\{H[i](x) \mid x \in \mathbf{a}\}$.

Estimation Each $\mathcal{S}[i]$ holds a tuple of size Thresh. The j'th entry of this tuple is the largest number of trailing zeros in any element of $H[i,j](\mathbf{a})$.

Algorithm 2 ChooseHashFunctions(n, Thresh, t)

```
1: switch AlgorithmType do
2:     case AlgorithmType==Bucketing
3:         H ← PickHashFunctions(H_Toeplitz(n, n), t)
4:     case AlgorithmType==Minimum
5:         H ← PickHashFunctions(H_Toeplitz(n, 3n), t)
6:     case AlgorithmType==Estimation
7:         s ← 10 log(1/ε)
8:         H ← PickHashFunctions(H_{s−wise}(n, n), t × Thresh)
   return H
```

Algorithm 3 ProcessUpdate$(\mathcal{S}, H, x, \mathsf{Thresh})$

```
1: for i ∈ [1, |H|] do
2:     switch AlgorithmType do
3:         case Bucketing
4:             m_i = S[i](0)
5:             if H[i]_{m_i}(x) == 0^{m_i} then
6:                 S[i](0) ← S[i](0) ∪ {x}
7:                 if size(S[i](0)) > Thresh then
8:                     S[i](1) ← S[i](1) + 1
9:                     for y ∈ S do
10:                        if H[i]_{m_i+1}(y) ≠ 0^{m_i+1} then
11:                            Remove(S[i](0), y)
12:        case Minimum
13:            if size(S[i]) < Thresh then
14:                S[i].Append(H[i](x))
15:            else
16:                j ← arg max(S[i])
17:                if S[i](j) > H[i](x) then
18:                    S[i](j) ← H[i](x)
19:        case Estimation
20:            for j ∈ [1, Thresh] do
21:                S[i, j] ← max(S[i, j], TrailZero(H[i, j](x)))
22: return S
```

3.1 A Recipe For Transformation

Observe that for each of the algorithms, the final computation of F_0 estimation depends on the sketch $\mathcal{S}$. Therefore, as long as for two streams $\mathbf{a}$ and $\hat{\mathbf{a}}$, if their corresponding sketches (and the hash functions chosen) match, then the three schemes presented above would return the same estimates.

The recipe for a transformation of streaming algorithms to model counting algorithms is based on the following insight:

1. Capture the relationship $\mathcal{P}(\mathcal{S}, H, \mathbf{a}_u)$ between the sketch $\mathcal{S}$, set of hash functions H, and set $\mathbf{a}_u$ at the end of stream.
2. View the formula φ as symbolic representation of the unique set $\mathbf{a}_u$ represented by the stream $\mathbf{a}$ such that $\mathsf{Sol}(\varphi) = \mathbf{a}_u$.
3. Given a formula φ and set of hash functions H, design an algorithm to construct sketch $\mathcal{S}$ such that the property $\mathcal{P}(\mathcal{S}, H, \mathsf{Sol}(\varphi))$ holds. Using the sketch $\mathcal{S}$, $|\mathsf{Sol}(\varphi)|$ can be estimated.

By applying the above recipe to the three F_0 estimation algorithms, we can derive corresponding model counting algorithms. In particular, applying the above recipe to

Algorithm 4 ComputeEst($\mathcal{S}$, Thresh)

switch AlgorithmType **do**
  case Bucketing
    return Median $\left(\left\{\text{size}(\mathcal{S}[i](0)) \times 2^{\mathcal{S}[i](1)}\right\}_i\right)$
  case Minimum
    return Median $\left(\left\{\frac{\mathsf{Thresh}\times 2^m}{\max\{\mathcal{S}[i]\}}\right\}_i\right)$
  case Estimation(r)
    return Median $\left(\left\{\frac{\ln\left(1-\frac{1}{\mathsf{Thresh}}\sum_{j=1}^{\mathsf{Thresh}} \mathbb{1}\{\mathcal{S}[i,j]\geq r\}\right)}{\ln(1-2^{-r})}\right\}_i\right)$

the Bucketing algorithm leads us to the state of the art hashing-based model counting algorithm, ApproxMC, proposed by Chakraborty et al. [5]. Applying the above recipe to Minimum and Estimation allows us to obtain different model counting schemes. In this extended abstract we illustrate this transformation for the Minimum-based algorithm.

3.2 Example Application of Recipe: Minimum-based Algorithm

We showcase the application of the recipe in the context of minimum-based algorithm. For a given multiset $\mathbf{a}$ (e.g.: a data stream or solutions to a model), we now specify the property $\mathcal{P}(\mathcal{S}, H, \mathbf{a}_u)$ as follows:

The sketch $\mathcal{S}$ is an array of sets indexed by members of H that holds lexicographically p minimum elements of $H[i](\mathbf{a}_u)$ where p is $\min(\frac{96}{\varepsilon^2}, |\mathbf{a}_u|)$. $\mathcal{P}$ is the property that specifies this relationship.

The following lemma due to Bar-Yossef et al. [2] establishes the relationship between the property $\mathcal{P}$ and the number of distinct elements of a multiset. Let $\max(S_i)$ denote the largest element of the set S_i.

LEMMA 1. *[2] Let $\mathbf{a} \subseteq \{0,1\}^n$ be a multiset. Let $H \subseteq \mathcal{H}_{\mathsf{Toeplitz}}(n, 3n)$ where each $H[i]$ is independently drawn from $\mathcal{H}_{\mathsf{Toeplitz}}(n, 3n)$, and $|H| = O(\log 1/\delta)$. Let $\mathcal{S}$ be such that $\mathcal{P}(\mathcal{S}, H, a_u)$ holds. Let $c = \text{Median}\ \{\frac{p\cdot 2^m}{\max(S[i])}\}_i$. Then*

$$\Pr\left[\frac{|\mathbf{a}_u|}{(1+\varepsilon)} \leq c \leq (1+\varepsilon)|\mathbf{a}_u|\right] \geq 1-\delta.$$

Therefore, we can transform the Minimum algorithm for F_0 estimation to that of model counting given access to a subroutine that can compute $\mathcal{S}$ such that $\mathcal{P}(\mathcal{S}, H, \mathsf{Sol}(\varphi))$ holds. The following proposition establishes the existence and complexity of such a subroutine, called FindMin.

PROPOSITION 1. *There is an algorithm* FindMin *that, given φ over n variables, $h \in \mathcal{H}_{\mathsf{Toeplitz}}(n, m)$, and p as input, returns a set, $\mathcal{B} \subseteq h(\mathsf{Sol}(\varphi))$ so that if $|h(\mathsf{Sol}(\varphi))| \leq p$, then $\mathcal{B} = h(\mathsf{Sol}(\varphi))$, otherwise $\mathcal{B}$ is the p lexicographically minimum elements of $h(\mathsf{Sol}(\varphi))$. Moreover, if φ is a CNF formula, then* FindMin *makes $\mathcal{O}(p \cdot m)$ calls to an NP oracle, and if φ is a DNF formula with k terms, then* FindMin *takes $\mathcal{O}(m^3 \cdot n \cdot k \cdot p)$ time.*

Equipped with Proposition 1, we are now ready to present the algorithm, called ApproxModelCountMin, for model counting. Since the complexity of FindMin is PTIME when φ is in DNF, we have ApproxModelCountMin as a FPRAS for DNF formulas.

Algorithm 5 ApproxModelCountMin($\varphi, \varepsilon, \delta$)

$t \leftarrow 35\log(1/\delta)$
$H \leftarrow \mathsf{PickHashFunctions}(\mathcal{H}_{\mathsf{Toeplitz}}(n, 3n), t)$
$S \leftarrow \{\}$
$\mathsf{Thresh} \leftarrow \frac{96}{\varepsilon^2}$
for $i \in [1, t]$ **do**
  $S[i] \leftarrow \mathsf{FindMin}(\varphi, H[i], \mathsf{Thresh})$
$Est \leftarrow \text{Median}\left(\left\{\frac{\mathsf{Thresh}\times 2^{3n}}{\max\{S[i]\}}\right\}_i\right)$
return Est

THEOREM 1. *Given φ, ε,δ,* ApproxModelCountMin *returns Est such that*

$$\Pr\left(\frac{|\mathsf{Sol}(\varphi)|}{1+\varepsilon} \leq Est \leq (1+\varepsilon)|\mathsf{Sol}(\varphi)|\right) \geq 1-\delta.$$

If φ is a CNF formula, then ApproxModelCountMin *is a polynomial-time algorithm that makes $\mathcal{O}(\frac{1}{\varepsilon^2} n \log(\frac{1}{\delta}))$ calls to an NP oracle. If φ is a DNF formula, then* ApproxModelCountMin *is an FPRAS.*

We now give a proof of Proposition 1 by describing the subroutine FindMin.

PROOF. We first present the algorithm when the formula φ is a DNF formula. Adapting the algorithm for the case of CNF can be done by using similar ideas. Let $\phi = T_1 \vee T_2 \vee \cdots \vee T_k$ be a DNF formula over n variables where T_i is a term. Let $h : \{0,1\}^n \to \{0,1\}^m$ be a linear hash function in $\mathcal{H}_{\mathsf{Toeplitz}}(n, m)$ defined by a $m \times n$ binary matrix A. Let $\mathcal{C}$ be the set of hashed values of the satisfying assignments for φ: $\mathcal{C} = \{h(x) \mid x \models \varphi\} \subseteq \{0,1\}^m$. Let $\mathcal{C}_p$ be the first p elements of $\mathcal{C}$ in the lexicographic order. Our goal is to compute $\mathcal{C}_p$.

We illustrate an algorithm with running time $O(m^3np)$ to compute $\mathcal{C}_p$ when the formula is just a term T. This algorithm can easily be generalized to formulas with k-terms. Let T be a term with width w (number of literals) and $\mathcal{C} = \{Ax \mid x \models T\}$. By fixing the variables in T we get a vector b_T and an $N \times (n-w)$ matrix A_T so that $\mathcal{C} = \{A_T x + b_T \mid x \in \{0,1\}^{(n-w)}\}$. Both A_T and b_T can be computed from A and T in linear time. Let $h_T(x)$ be the transformation $A_T x + b_T$.

We will compute $\mathcal{C}_p$ iteratively as follows: assuming we have computed the $(q-1)^{th}$ minimum of $\mathcal{C}$, we will compute the q^{th} minimum using a prefix-search strategy. We will use a subroutine to solve the following basic prefix-search primitive: Given any l bit string $y_1 \ldots y_l$, is there an $x \in \{0,1\}^{n-w}$ so that $y_1 \ldots y_l$ is a prefix for some string in $\{h_T(x)\}$? This task can be performed using Gaussian elimination over an $(l+1) \times (n-w)$ binary matrix and can be implemented in time $O(l^2(n-w))$.

Let $y = y_1 \ldots y_m$ be the $(q-1)^{th}$ minimum in $\mathcal{C}$. Let r_1 be the rightmost 0 of y. Then using the above-mentioned procedure we can find the lexicographically smallest string in the range of h_T that extends $y_1 \ldots y_{(r-1)}1$ if it exists. If no such string exists in $\mathcal{C}$, find the index of the next 0 in y and repeat the procedure. In this manner the q^{th} minimum can be computed using $O(m)$ calls to the prefix-searching primitive resulting in an $O(m^3 n)$ time algorithm. Invoking the above procedure p times results in an algorithm to compute $\mathcal{C}_p$ in $O(m^3np)$ time. □

3.3 Distributed DNF Counting

Consider the problem of *distributed DNF counting*. In this setting, there are k sites that can each communicate with a central coordinator. The input DNF formula φ is partitioned into k DNF subformulas $\varphi_1, \ldots, \varphi_k$, where each φ_i is a subset of the terms of the original φ, with the j'th site receiving only φ_j. The goal is for the coordinator to obtain an (ϵ, δ)-approximation of the number of solutions to φ, while minimizing the total number of bits communicated between the sites and the coordinator. Distributed algorithms for sampling and counting solutions to CSP's have been studied recently in other models of distributed computation [12, 11, 13, 14]. From a practical perspective, given the centrality of #DNF in the context of probabilistic databases [28, 27], a distributed DNF counting algorithm would entail applications in distributed probabilistic databases.

From our perspective, distributed DNF counting falls within the *distributed functional monitoring* framework formalized by Cormode et al. [7]. Here, the input is a stream $\mathbf{a}$ which is partitioned arbitrarily into sub-streams $\mathbf{a}_1, \ldots, \mathbf{a}_k$ that arrive at each of k sites. Each site can communicate with the central coordinator, and the goal is for the coordinator to compute a function of the joint stream $\mathbf{a}$ while minimizing the total communication. This general framework has several direct applications and has been studied extensively (see [8, 18, 33] and the references therein).

In distributed DNF counting problem, each sub-stream $\mathbf{a}_i$ corresponds to the set of satisfying assignments to each subformula φ_i, while the function to be computed is F_0.

The algorithms discussed in Section 3 can be extended to the distributed setting. We briefly describe the distributed implementation of the minimum based algorithm.

Distributed implementation of the minimum-based algorithm. The coordinator chooses hash functions $H[1], \ldots, H[t]$ from $\mathcal{H}_{\mathsf{Toeplitz}}(n, 3n)$ and sends them to the k sites. Each site runs the FindMin algorithm for each hash function and sends the outputs to the coordinator. So, the coordinator receives sets $S[i, j]$, consisting of the Thresh lexicographically smallest hash values of the solutions to φ_j. The coordinator then extracts $S[i]$, the Thresh lexicographically smallest elements of $S[i, 1] \cup \cdots \cup S[i, k]$ and proceeds with the rest of algorithm ApproxModelCountMin. The communication cost is $O(kn/\varepsilon^2 \cdot \log(1/\delta))$ to account for the k sites sending the outputs of their FindMin invocations. The time complexity for each site is polynomial in n, ε^{-1}, and $\log(\delta^{-1})$.

A straightforward implementation of the Bucketing algorithm leads to a distributed DNF counting algorithm whose communication cost is $\tilde{O}(k(n + 1/\varepsilon^2) \cdot \log(1/\delta))$ and time complexity per site is polynomial in n, ε^{-1}, and $\log(\delta^{-1})$. Similarly, the estimation based algorithm leads to a distributed algorithm with $\tilde{O}(k(n + 1/\varepsilon^2) \log(1/\delta))$ communication cost. However, we do not know a polynomial time algorithm to implement the last algorithm on DNF terms.

Lower Bound

The communication cost for the Bucketing and Estimation-based algorithms is nearly optimal in their dependence on k and ε. Woodruff and Zhang [33] showed that the randomized communication complexity of estimating F_0 up to a $1 + \varepsilon$ factor in the distributed functional monitoring setting is $\Omega(k/\varepsilon^2)$. We can reduce the F_0 estimation problem to distributed DNF counting. Namely, if for the F_0 estimation problem, the j'th site receives items $a_1, \ldots, a_m \in [N]$, then for the distributed DNF counting problem, φ_j is a DNF formula on $\lceil \log_2 N \rceil$ variables whose solutions are exactly $a_1, \ldots, a_m$ in their binary encoding. Thus, we immediately get an $\Omega(k/\varepsilon^2)$ lower bound for the distributed DNF counting problem. Finding the optimal dependence on N for $k > 1$ remains an interesting open question.

4. FROM COUNTING TO STREAMING

In this section we consider the *structured set streaming model* where each item S_i of the stream is a succinct representation of a set over the universe $U = \{0, 1\}^n$. Our goal is to design efficient algorithms (both in terms of memory and processing time per item) for computing $|\cup_i S_i|$ - the number of distinct elements in the union of all the sets in the stream. We call this problem F_0 computation over structured set streams. We discuss two types of structured sets *DNF Sets* and *Affine Spaces*. As we mentioned in the introduction, other structured sets studied in the literature are single and multi-dimensional ranges. Our techniques also give algorithms for estimating F_0 of such structured set streams, which we omit in this extended abstract.

DNF Sets

A particular representation we are interested in is where each set is presented as the set of satisfying assignments to a DNF formula. Let φ be a DNF formula over n variables. Then the DNF set corresponding to φ be the set of satisfying assignments of φ. The *size* of this representation is the number of terms in the formula φ. A stream over DNF sets is a stream of DNF formulas $\varphi_1, \varphi_2, \ldots$. Given such a DNF stream, the goal is to estimate $|\bigcup_i S_i|$ where S_i the DNF set represented by φ_i. This quantity is the same as the number of satisfying assignments of the formula $\vee_i \varphi_i$. We show that the algorithms described in the previous section carry over to obtain (ϵ, δ) estimation algorithms for this problem with space and per-item time $\mathrm{poly}(1/\epsilon, n, k, \log(1/\delta))$ where k is the size of the formula.

THEOREM 2. *There is a streaming algorithm to compute an (ϵ, δ) approximation of F_0 over DNF sets. This algorithm takes space $O(\frac{n}{\varepsilon^2} \cdot \log \frac{1}{\delta})$ and processing time $O(n^4 \cdot k \cdot \frac{1}{\varepsilon^2} \cdot \log \frac{1}{\delta})$ per item where k is the size (number of terms) of the corresponding DNF formula.*

PROOF. We show how to adapt the Minimum-value based algorithm from Section 3.2 to this setting. The algorithm picks a hash function $h \in \mathcal{H}_{\mathsf{Toeplitz}}(n, 3n)$ and maintains the set $\mathcal{B}$ consisting of t lexicographically minimum elements of the set $\{h(\mathsf{Sol}(\varphi_1 \vee \ldots \vee \varphi_{i-1}))\}$ after processing $i - 1$ items. When φ_i arrives, it computes the set $\mathcal{B}'$ consisting of the t lexicographically minimum values of the set $\{h(\mathsf{Sol}(\varphi_i))\}$ and subsequently updates $\mathcal{B}$ by computing the t lexicographically smallest elements from $\mathcal{B} \cup \mathcal{B}'$. By Proposition 1, the computation of $\mathcal{B}'$ can be done in time $O(n^4 \cdot k \cdot t)$ where k is the number of terms in φ_i. Updating $\mathcal{B}$ can be done in $O(t \cdot n)$ time. Thus, the update time for the item φ_i is $O(n^4 \cdot k \cdot t)$. For obtaining an (ε, δ)-approximation, we set $t = O(\frac{1}{\varepsilon^2})$ and repeat the procedure $O(\log \frac{1}{\delta})$ times and take the median value. Thus the update time for item φ is $O(n^4 \cdot k \cdot \frac{1}{\varepsilon^2} \cdot \log \frac{1}{\delta})$. For analyzing space, each hash function uses $O(n)$ bits and and the algorithm stores $O(\frac{1}{\epsilon^2})$ mini-

mums, resulting in overall space usage of $O(\frac{n}{\varepsilon^2} \cdot \log \frac{1}{\delta})$. The proof of correctness follows from Lemma 1. □

Instead of the Minimum-value based algorithm, we could also adapt the Bucketing-based algorithm to obtain an algorithm with similar space and time complexities.

Affine Spaces

Another example of a structured stream is where each item of the stream is an affine space represented by $Ax = B$ where A is a Boolean matrix and B is a zero-one vector. Without loss of generality, we may assume that A is a $n \times n$ matrix. Thus an affine stream consists of $\langle A_1, B_1\rangle, \langle A_2, B_2\rangle, \cdots$, where each $\langle A_i, B_i\rangle$ succinctly represents a set $\{x \in \{0,1\}^n \mid A_i x = B_i\}$. Here operations are over the finite field of size 2. For a $n \times n$ Boolean matrix A and a zero-one vector B, let $\mathsf{Sol}(\langle A, B\rangle)$ denote the set of all x that satisfy $Ax = B$.

PROPOSITION 2. *Given (A, B), $h \in \mathcal{H}_{\mathsf{Toeplitz}}(n, 3n)$, and t as input, there is an algorithm,* AffineFindMin, *that returns a set, $\mathcal{B} \subseteq h(\mathsf{Sol}(\langle A, B\rangle))$ so that if $|h(\mathsf{Sol}(\langle A, B\rangle))| \leq t$, then $\mathcal{B} = h(\mathsf{Sol}(\langle A, B\rangle))$, otherwise $\mathcal{B}$ is the t lexicographically minimum elements of $h(\mathsf{Sol}(\langle A, B\rangle))$. The time taken by this algorithm is $O(n^4 t)$ and the space taken by the algorithm is $O(tn)$.*

The above proposition together with the minimum-based algorithm gives the following theorem.

THEOREM 3. *There is a streaming algorithm that computes a (ϵ, δ)-approximation of F_0 over affine spaces. This algorithm takes space $O(\frac{n}{\epsilon^2} \cdot \log(1/\delta))$ and processing time of $O(n^4 \frac{1}{\epsilon^2} \log(1/\delta))$ per item.*

5. RELATING SKETCH SPACE COMPLEXITY AND NP QUERY COMPLEXITY

Our investigations reveal surprising connections between algorithms for F_0 estimation and model counting that are of interest to two different communities.

It is noteworthy that the two communities often have different concerns: in the context of model counting, one is focused on the NP-query complexity while in the context of streaming, the focus is on the space complexity. This begs the question of whether the connections are a matter of happenstance or there is an inherent relationship between the space complexity in the context of streaming and the query complexity for model counting. We detail our investigations on the existence of such a relationship.

In the following, we will fold the hash function h also in the sketch S. With this simplification, instead of writing $P(S, h, \mathsf{Sol}(\varphi))$ we write $P(S, \mathsf{Sol}(\varphi))$.

We first introduce some complexity-theoretic notation. For a complexity class $\mathcal{C}$, a language L belongs to the complexity class $\exists \cdot \mathcal{C}$ if there is a polynomial $q(\cdot)$ and a language $L' \in \mathcal{C}$ such that for every x, $x \in L \Leftrightarrow \exists y, |y| \leq q(|x|), \langle x, y\rangle \in L'$.

Consider a streaming algorithm for F_0 that constructs a sketch such that $P(S, a_u)$ holds for some property P using which we can estimate $|a_u|$, where the size of S is polylogarithmic in the size of the universe and polynomial in $1/\varepsilon$. Now consider the following *Sketch-Language*

$$L_{sketch} = \{\langle \varphi, S\rangle \mid P(S, \mathsf{Sol}(\varphi)) \text{ holds}\}.$$

THEOREM 4. *If L_{sketch} belongs to the complexity class $\mathcal{C}$, then there exists a $\mathrm{FP}^{\exists \cdot \mathcal{C}}$ model counting algorithm that estimates the number of satisfying assignments of a given formula φ. The number of queries made by the algorithm is bounded by the sketch size.*

Let us apply the above theorem to the minimum-based algorithm. The sketch language consists of tuples of the form $\langle \varphi, \langle h, v_1, \cdots, v_t\rangle\rangle$ where $\{v_1, \cdots v_t\}$ is the set of t lexicographically smallest elements of the set $h(\mathsf{Sol}(\varphi))$. It can be see that this language is in coNP. Since $\exists \cdot \text{coNP}$ is same as the class Σ_2^{P}, we obtain a $\mathrm{FP}^{\Sigma_2^{\mathrm{P}}}$ algorithm. Since $t = O(1/\varepsilon^2)$ and h maps from n-bit strings to $3n$-bit strings, it follows that the size of the sketch is $O(n/\varepsilon^2)$. Thus the number of queries made by the algorithm is $O(n/\varepsilon^2)$.

Interestingly, all the model counting algorithms that were obtained following our recipe are probabilistic polynomial-time algorithms that make queries to languages in NP. The above generic transformation gives a deterministic polynomial-time algorithm that makes queries to a Σ_2^{P} language. Precisely characterizing the properties of the sketch that lead to probabilistic algorithms making only NP queries is an interesting direction to explore.

6. CONCLUSION AND FUTURE OUTLOOK

Our investigation led to a diverse set of results that unify over two decades of work in model counting and F_0 estimation. The viewpoint presented in this work has the potential to spur several new interesting research directions.

Higher Moments. There has been a long line of work on estimation of higher moments, i.e. F_k over data streams. A natural direction of future research is to adapt the notion of F_k in the context of model of counting and explore its applications. We expect extensions of the framework and recipe presented in this work to derive algorithms for higher frequency moments in the context of model counting.

Sparse XORs. In the context of model counting, the performance of underlying SAT solvers strongly depends on the size of XORs. The standard constructions lead to XORs of size $\Theta(n)$ and an interesting line of research has focused on the design of sparse XOR-based hash functions [17, 19, 10] culminating in showing that one can use hash functions of the form $h(x) = Ax + b$ wherein each entry of the m-th row of A is 1 with probability $\mathcal{O}(\frac{\log m}{m})$ [23]. Such XORs were shown to improve the runtime efficiency. In this context, a natural direction would be to explore the usage of sparse XORs in the context of F_0 estimation.

Acknowledgements

We thank the anonymous reviewers of PODS 21 for valuable comments. We are grateful to Phokion Kolaitis for suggesting exploration beyond the transformation recipe that led to results in Section 5. Bhattacharyya was supported in part by the NRF Fellowship Programme [NRF-NRFFAI1-2019-0002] and an Amazon Research Award. Meel was supported in part by the NRF Fellowship Programme[NRF-NRFFAI1-2019-0004] and the AI Singapore Programme [AISG-RP-2018-005], and NUS ODPRT Grant [R-252-000-685-13]. Vinod was supported in part by NSF CCF-2130608, NSF CCF-184908 and NSF HDR:TRIPODS-1934884 awards. Pavan was supported in part by NSF CCF-2130536, NSF CCF-1849053 and NSF HDR:TRIPODS-1934884 awards.

7. REFERENCES

[1] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. *J. Comput. Syst. Sci.*, 58(1):137–147, 1999.

[2] Z. Bar-Yossef, T. S. Jayram, R. Kumar, D. Sivakumar, and L. Trevisan. Counting distinct elements in a data stream. In *Proc. of RANDOM*, volume 2483, pages 1–10, 2002.

[3] Z. Bar-Yossef, R. Kumar, and D. Sivakumar. Reductions in streaming algorithms, with an application to counting triangles in graphs. In *Proc. of SODA*, pages 623–632. ACM/SIAM, 2002.

[4] J. L. Carter and M. N. Wegman. Universal classes of hash functions. In *Proceedings of the ninth annual ACM symposium on Theory of computing*, pages 106–112. ACM, 1977.

[5] S. Chakraborty, K. S. Meel, and M. Y. Vardi. Algorithmic improvements in approximate counting for probabilistic inference: From linear to logarithmic SAT calls. In *Proc. of IJCAI*, 2016.

[6] G. Cormode and S. Muthukrishnan. Estimating dominance norms of multiple data streams. In G. D. Battista and U. Zwick, editors, *Proc. of ESA*, volume 2832 of *Lecture Notes in Computer Science*, pages 148–160. Springer, 2003.

[7] G. Cormode, S. Muthukrishnan, and K. Yi. Algorithms for distributed functional monitoring. *ACM Transactions on Algorithms (TALG)*, 7(2):1–20, 2011.

[8] G. Cormode, S. Muthukrishnan, K. Yi, and Q. Zhang. Continuous sampling from distributed streams. *Journal of the ACM (JACM)*, 59(2):1–25, 2012.

[9] P. Dagum, R. Karp, M. Luby, and S. Ross. An optimal algorithm for monte carlo estimation. *SIAM Journal on computing*, 29(5):1484–1496, 2000.

[10] S. Ermon, C. P. Gomes, A. Sabharwal, and B. Selman. Low-density parity constraints for hashing-based discrete integration. In *Proc. of ICML*, pages 271–279, 2014.

[11] W. Feng, T. P. Hayes, and Y. Yin. Distributed symmetry breaking in sampling (optimal distributed randomly coloring with fewer colors). *arXiv preprint arXiv:1802.06953*, 2018.

[12] W. Feng, Y. Sun, and Y. Yin. What can be sampled locally? *Distributed Computing*, pages 1–27, 2018.

[13] W. Feng and Y. Yin. On local distributed sampling and counting. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*, pages 189–198, 2018.

[14] M. Fischer and M. Ghaffari. A simple parallel and distributed sampling technique: Local glauber dynamics. In *32nd International Symposium on Distributed Computing*, 2018.

[15] P. Flajolet and G. N. Martin. Probabilistic counting algorithms for data base applications. *J. Comput. Syst. Sci.*, 31(2):182–209, 1985.

[16] P. B. Gibbons and S. Tirthapura. Estimating simple functions on the union of data streams. In A. L. Rosenberg, editor, *Proc. of SPAA*, pages 281–291. ACM, 2001.

[17] C. P. Gomes, J. Hoffmann, A. Sabharwal, and B. Selman. From sampling to model counting. In *Proc. of IJCAI*, pages 2293–2299, 2007.

[18] Z. Huang, K. Yi, and Q. Zhang. Randomized algorithms for tracking distributed count, frequencies, and ranks. In *Proc. of PODS*, pages 295–306, 2012.

[19] A. Ivrii, S. Malik, K. S. Meel, and M. Y. Vardi. On computing minimal independent support and its applications to sampling and counting. *Constraints*, pages 1–18, 2015.

[20] D. M. Kane, J. Nelson, and D. P. Woodruff. An optimal algorithm for the distinct elements problem. In *Proc. of PODS*, pages 41–52. ACM, 2010.

[21] R. Karp and M. Luby. Monte-carlo algorithms for enumeration and reliability problems. *Proc. of FOCS*, 1983.

[22] R. M. Karp, M. Luby, and N. Madras. Monte-carlo approximation algorithms for enumeration problems. *Journal of Algorithms*, 10(3):429 – 448, 1989.

[23] K. S. Meel and S. Akshay. Sparse hashing for scalable approximate model counting: Theory and practice. In *Proc. of LICS*, 2020.

[24] K. S. Meel, A. A. Shrotri, and M. Y. Vardi. On hashing-based approaches to approximate dnf-counting. In *In Proc. of FSTTCS*, 2017.

[25] K. S. Meel, A. A. Shrotri, and M. Y. Vardi. Not all fprass are equal: Demystifying fprass for dnf-counting (extended abstract). In *Proc. of IJCAI*, 8 2019.

[26] A. Pavan and S. Tirthapura. Range-efficient counting of distinct elements in a massive data stream. *SIAM J. Comput.*, 37(2):359–379, 2007.

[27] C. Ré and D. Suciu. Approximate lineage for probabilistic databases. *Proceedings of the VLDB Endowment*, 1(1):797–808, 2008.

[28] P. Senellart. Provenance and probabilities in relational databases. *ACM SIGMOD Record*, 46(4):5–15, 2018.

[29] M. Soos and K. S. Meel. Bird: Engineering an efficient cnf-xor sat solver and its applications to approximate model counting. In *Proceedings of AAAI Conference on Artificial Intelligence (AAAI)(1 2019)*, 2019.

[30] L. Stockmeyer. The complexity of approximate counting. In *Proc. of STOC*, pages 118–126, 1983.

[31] S. Tirthapura and D. P. Woodruff. Rectangle-efficient aggregation in spatial data streams. In *Proc. of PODS*, pages 283–294. ACM, 2012.

[32] L. Valiant. The complexity of enumeration and reliability problems. *SIAM Journal on Computing*, 8(3):410–421, 1979.

[33] D. P. Woodruff and Q. Zhang. Tight bounds for distributed functional monitoring. In *Proceedings of the forty-fourth annual ACM symposium on Theory of computing*, pages 941–960, 2012.