SIGMOD Officers, Committees, and Awardees

SIGMOD Edgar F. Codd Innovations Award

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

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

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)

SIGMOD Contributions Award

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

Maria Zemankova (1992)	Gio Wiederhold (1995)	Yahiko Kambayashi (1995)
Jeffrey Ullman (1996)	Avi Silberschatz (1997)	Won Kim (1998)
Raghu Ramakrishnan (1999)	Michael Carey (2000)	Laura Haas (2000)
Daniel Rosenkrantz (2001)	Richard Snodgrass (2002)	Michael Ley (2003)
Surajit Chaudhuri (2004)	Hongjun Lu (2005)	Tamer Özsu (2006)
Hans-Jörg Schek (2007)	Klaus R. Dittrich (2008)	Beng Chin Ooi (2009)
David Lomet (2010)	Gerhard Weikum (2011)	Marianne Winslett (2012)
H.V. Jagadish (2013)	Kyu-Young Whang (2014)	Curtis Dyreson (2015)
Samuel Madden (2016)	Yannis E. Ioannidis (2017)	Z. Meral Özsoyoğlu (2018)
Ahmed Elmagarmid (2019)		

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

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

[Last updated: December 31, 2019]

Guest Editor's Notes

Welcome to the March 2020 issue of the ACM SIGMOD Record!

The new year of 2020 begins with a special issue on the **2019 ACM SIGMOD Research Highlight Award**. This is an award for the database community to showcase a set of research projects that exemplify core database research. In particular, these projects address an important problem, represent a definitive milestone in solving the problem, and have the potential of significant impact. This award also aims to make the selected works widely known in the database community, to our industry partners, and to the broader ACM community.

The award committee and editorial board included Rada Chirkova, Wim Martens, Jun Yang, and Divesh Srivastava. We solicited articles from PODS 2019, SIGMOD 2019, VLDB 2019, ICDE 2019, EDBT 2019, and ICDT 2019, as well as from community nominations. Through a careful review process eight articles were finally selected as 2019 Research Highlights. The authors of each article worked closely with an associate editor to rewrite the article into a compact 8-page format, and improved it to appeal to the broad data management community. In addition, each research highlight is accompanied by a one-page technical perspective written by an expert on the topic presented in the article. The technical perspective provides the reader with an overview of the background, the motivation, and the key innovation of the featured research highlight, as well as its scientific and practical significance.

The 2019 research highlights cover a broad set of topics, including (a) an automatic way of checking for invariant confluence to enable scaling distributed database systems with consistent semantics ("Checking Invariant Confluence, In Whole or In Parts"); (b) a checkpoint and recovery method with the ability to scale throughput linearly on a large multicore server with negligible increase of latency ("Concurrent Prefix Recovery: Performing CPR on a Database"); (c) the computational complexity of regular document spanners, which provide an abstraction for Information Extraction rules ("Constant-Delay Enumeration for Nondeterministic Document Spanners"); (d) a database perspective on the problem of fairness in machine learning ("Database Repair Meets Algorithmic Fairness"); (e) showing the potential of recursive computations on an RDBMS as the backend for large-scale machine learning ("Declarative Recursive Computation on an RDBMS"); (f) a general framework for detecting if all three problems of enumeration, counting and uniform generation are efficiently solvable ("Efficient Logspace Classes for Enumeration, Counting, and Uniform Generation"); (g) the use of core RDMBS techniques to explain the predictions of a deep ML model ("Query Optimization for Faster Deep CNN Explanations"); and (h) an efficient and tamper-proof way to retrieve and use historical data on a blockchain ("Revealing Every Story of Data in Blockchain Systems").

On behalf of the SIGMOD Record Editorial Board, I hope that you enjoy reading the March 2020 issue of the SIGMOD Record!

Divesh Srivastava

March 2020

Your submissions to the SIGMOD Record are welcome via the submission site:
https://mc.manuscriptcentral.com/sigmodrecord

Prior to submission, please read the Editorial Policy on the SIGMOD Record's website:
https://sigmodrecord.org/sigmod-record-editorial-policy/

Past SIGMOD Record Editors:

Yanlei Diao (2014-2019)	Ioana Manolescu (2009-2013)	Alexandros Labrinidis (2007–2009)
Mario Nascimento (2005–2007)	Ling Liu (2000–2004)	Michael Franklin (1996–2000)
Jennifer Widom (1995–1996)	Arie Segev (1989–1995)	Margaret H. Dunham (1986–1988)
Jon D. Clark (1984–1985)	Thomas J. Cook (1981–1983)	Douglas S. Kerr (1976-1978)
Randall Rustin (1974-1975)	Daniel O'Connell (1971–1973)	Harrison R. Morse (1969)

TECHNICAL PERSPECTIVE: Checking Invariant Confluence, In Whole or In Parts

Johannes Gehrke
Microsoft Research
Redmond, WA; USA
johannes@acm.org

Never make a promise - you may have to keep it. — Neil Jordan

Database systems were known to provide strong consistency guarantees. As an example, database textbook defines the ACID guarantees as "four important properties of transactions to maintain data in the face of concurrent access and system failures" [2]. Beyond atomicity, consistency, and durability, the "I" in ACID is loosely defined as "Users should be able to understand a transaction without considering the effects of other concurrently executing transactions, even if the DBMS interleaves the actions of several transactions for performance reasons" [2]. In the resulting model called serializable execution, each transaction operates like it has the database to itself, and the result of running a set of transactions is equivalent to some serial execution of these transactions.

Serializable execution implies an ordering of transactions based on conflicts, where a conflict means that we have two transactions that are accessing the same database record, and at least one of them is a write. To ensure a serial ordering of the transactions, the conflicts must be totally ordered across transactions. In other words, if two transactions conflict, they need to coordinate. However, coordination has a price. If the database system is located within a single data center, coordination across nodes costs a few hundred microseconds, but if the system is wide-area distributed across several data centers, the delay between nodes may be in the 10s of milliseconds, restricting throughput to 10s of operations in the worst case. And in case the network is partitioned, the system may not be available at all.

For a while, most distributed systems dealt with this tradeoff in one of two ways. One popular option was to focus on high availability and low latency and perform the coordination asynchronously. Unfortunately this approach only provides weak consistency guarantees, so applications must use additional mechanisms such as compensation transactions or custom conflict resolution strategies, or they must restrict the programming model to eliminate the possibility of conflicts, for example by only allowing updates of a single record. Another approach was to insist on strong consistency and to accept slower response times because of coordination between nodes. It seemed like consistent semantics was not such a good idea after all.

In a recent landmark paper, Bailis et al asked the intriguing question: Are there cases where we may not need coordination between transactions at all, and thus we can achieve both high availability and low latency while maintaining application-level constraints [1]? The somewhat surprising answer is that in many practical scenarios the answer is yes. The paper introduced *invariant confluence*, a criterion that determines whether a set of transactions requires coordination for correct execution while maintaining integrity constraints. The framework requires developers to state the integrity constraints of the application on the database state a priori, but then it provides a necessary and sufficient condition for coordination-free execution. The resulting system only needs to coordinate in cases where the framework indicates that coordination is necessary; if it is possible, the framework guarantees that transactions do not violate any of the stated integrity constraints even if transactions do not coordinate. Analyzing TPC-C with invariant confluence by manually extracting the inherent constraints enabled Bailis et al. to show that only two of the twelve TPC-C constraints are not invariant confluent, and when applying the resulting insights to scaling TPC-C, they outperformed the previous best result of scaling TPC-C New-Order performance by factor of 25! However, coming up with the constraints and analyzing them is challenging as the authors admit themselves in the paper: "We have found the process of invariant specification to be non-trivial but feasible in practice;" [1].

The following paper is automating this manual process. The task of determining invariant confluence for an object given a set of transactions is no longer an exercise for the reader; instead, the paper provides an automatic way of checking for invariant confluence — a leap forward towards making the concept practical. The paper also goes a step further by taking objects that are not invariant confluent and in some cases allowing at least only occasional coordination instead of requiring coordination all the time. A beautiful set of results that is an important step towards scaling distributed database systems — with consistent semantics after all.

1. REFERENCES

[1] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Coordination avoidance in database systems. *Proc. VLDB Endow.*, 8(3):185–196, 2014.

[2] R. Ramakrishnan and J. Gehrke. *Database management systems (3. ed.)*. McGraw-Hill, 2003.

Checking Invariant Confluence, In Whole or In Parts

Michael Whittaker
UC Berkeley
Berkeley, CA
mjwhittaker@berkeley.edu

Joseph M. Hellerstein
UC Berkeley
Berkeley, CA
hellerstein@berkeley.edu

ABSTRACT

Strongly consistent distributed systems are easy to reason about but face fundamental limitations in availability and performance. Weakly consistent systems can be implemented with very high performance but place a burden on the application developer to reason about complex interleavings of execution. Invariant confluence provides a formal framework for understanding when we can get the best of both worlds. An invariant confluent object can be efficiently replicated with no coordination needed to preserve its invariants. However, actually determining whether or not an object is invariant confluent is challenging.

In this paper, we establish conditions under which a commonly used sufficient condition for invariant confluence is both necessary and sufficient, and we use this condition to design a general-purpose interactive invariant confluence decision procedure. We then take a step beyond invariant confluence and introduce a generalization of invariant confluence, called segmented invariant confluence, that allows us to replicate non-invariant confluent objects with a small amount of coordination. We implement these formalisms in a prototype called Lucy and find that our decision procedures efficiently handle common real-world workloads including foreign keys, escrow transactions, and more.

1. INTRODUCTION

When an application designer decides to replicate a piece of data, they have to make a fundamental choice between weak and strong consistency. Replicating the data with strong consistency using a technique like distributed transactions [7] or state machine replication [14] makes the application designer's life very easy. To the developer, a strongly consistent system behaves exactly like a single-threaded system running on a single node, so reasoning about the behavior of the system is simple [12]. Unfortunately, strong consistency is at odds with performance. The CAP theorem and PACELC theorem tell us that strongly consistent systems suffer from higher latency at best and unavailability at worst [9, 1]. On the other hand, weak consistency models like eventual consistency [24], PRAM consistency [17], causal consistency [2], and others [19, 20] allow data to be replicated with high availability and low latency, but they put a tremendous burden on the application designer to reason about the complex interleavings of operations that are allowed by these weak consistency models. In particular, weak consistency models strip an application developer of one of the earliest and most effective tools that is used to reason about the execution of programs: application invariants [13, 5] such as database integrity constraints [11]. Even if every transaction executing in a weakly consistent system individually maintains an application invariant, the system as a whole can produce invariant-violating states.

Is it possible for us to have our strongly consistent cake and eat it with high availability too? Can we replicate a piece of data with weak consistency but still ensure that its invariants are maintained? Yes... sometimes. Bailis et al. introduced the notion of *invariant confluence* as a necessary and sufficient condition for when invariants can be maintained over replicated data without the need for any coordination [3]. If an object is invariant confluent with respect to an invariant, we can replicate it with the performance benefits of weak consistency and (some of) the correctness benefits of strong consistency.

Unfortunately, to date, the task of identifying whether or not an object actually is invariant confluent has remained an exercise in human proof generation. Bailis et al. manually categorized a set of common objects, transactions, and invariants (e.g. foreign key constraints on relations, linear constraints on integers) as invariant confluent or not. Hand-written proofs of this sort are unreasonable to expect from programmers. Ideally we would have a general-purpose program that could automatically determine invariant confluence for us. **The first main thrust of this paper is to make invariant confluence checkable:** to design a general-purpose invariant confluence decision procedure, and implement it in an interactive system.

Unfortunately, designing such a general-purpose decision procedure is impossible because determining the invariant confluence of an object is undecidable in general. Still, we can develop a decision procedure that works well in the common case. For example, many prior efforts have developed decision procedures for *invariant closure*, a sufficient (but not necessary) condition for invariant confluence [16, 15]. The existing approaches check whether an object is invari-

The original version of this paper is entitled "Interactive Checks for Coordination Avoidance" and was published in PVLDB Vol. 12.

ant closed. If it is, then they conclude that it is also invariant confluent. If it's not, then the current approaches are unable to conclude anything about whether or not the object is invariant confluent.

In this paper, we take a step back and study the underlying reason *why* invariant closure is not necessary for invariant confluence. Using this understanding, we construct a set of modest conditions under which invariant closure and invariant confluence are in fact *equivalent*, allowing us to reduce the problem of determining invariant confluence to that of determining invariant closure. Then, we use these conditions to design a general-purpose interactive invariant confluence decision procedure.

The second main thrust of this paper is to partially avoid coordination even in programs that require it, by generalizing invariant confluence to a property called *segmented invariant confluence.* While invariant confluence characterizes objects that can be replicated *without any* coordination, segmented invariant confluence allows us to replicate non-invariant confluent objects with only *occasional* coordination. The main idea is to divide the set of invariant-satisfying states into *segments*, with a restricted set of transactions allowed in each segment. Within a segment, servers act without any coordination; they synchronize only to transition across segment boundaries. This design highlights the trade-off between application complexity and coordination-freedom; more complex applications require more segments which require more coordination.

Finally, we present Lucy: an implementation of our decision procedures and a system for replicating invariant confluent and segmented invariant confluent objects. Using Lucy, we find that our decision procedures can efficiently handle a wide range of common workloads. For example, in Section 6, we apply Lucy to foreign key constraints and escrow transactions. Lucy processes these workloads in less than half a second, and no workload requires more than 66 lines of code to specify.

2. INVARIANT CONFLUENCE

Informally, a replicated object is **invariant confluent** with respect to an invariant if every replica of the object is guaranteed to satisfy the invariant despite the possibility of different replicas being concurrently modified or merged together [3]. In this section, we make this informal notion of invariant confluence precise.

We begin by introducing our system model of replicated objects in which a distributed object and an invariant are replicated across a set of servers. Clients send transactions to servers, and servers execute transactions so long as they maintain the invariant. Servers execute transactions without coordination, but to avoid state divergence, servers periodically gossip with one another and merge their replicas.

2.1 System Model

A **distributed object** $O = (S, \sqcup)$ consists of a set S of states and a binary merge operator $\sqcup : S \times S \to S$ that merges two states into one. A **transaction** $t : S \to S$ is a function that maps one state to another. An **invariant** I is a subset of S. Notationally, we write $I(s)$ to denote that s satisfies the invariant (i.e. $s \in I$) and $\neg I(s)$ to denote that s does not satisfy the invariant (i.e. $s \notin I$).

Example 1. $O = (\mathbb{Z}, \max)$ is a distributed object consisting of integers merged by the max function; $t(x) = x + 1$ is a transaction that adds one to a state; and $\{x \in \mathbb{Z} \,|\, x \geq 0\}$ is the invariant that states x are non-negative.

In our system model, a distributed object O is replicated across a set $p_1, \ldots, p_n$ of n servers. Each server p_i manages a replica $s_i \in S$ of the object. Every server begins with a start state $s_0 \in S$, a fixed set T of transactions, and an invariant I. Servers repeatedly perform one of two actions.

First, a client can contact a server p_i and request that it execute a transaction $t \in T$. p_i speculatively executes t, transitioning from state s_i to state $t(s_i)$. If $t(s_i)$ satisfies the invariant—i.e. $I(t(s_i))$—then p_i commits the transaction and remains in state $t(s_i)$. Otherwise, p_i aborts the transaction and reverts to state s_i.

Second, a server p_i can send its state s_i to another server p_j with state s_j causing p_j to transition from state s_j to state $s_i \sqcup s_j$. Servers periodically merge states with one another in order to keep their states loosely synchronized. Note that unlike with transactions, servers *cannot* abort a merge; server p_j must transition from s_j to $s_i \sqcup s_j$ whether or not $s_i \sqcup s_j$ satisfies the invariant.

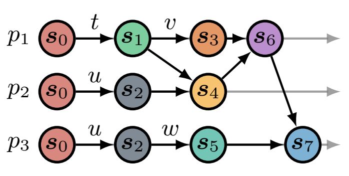
Informally, O is **invariant confluent** with respect to s_0, T, and I, abbreviated (s_0, T, I)**-confluent**, if every replica $s_1, \ldots, s_n$ is guaranteed to always satisfy the invariant I in every possible execution of the system.

2.2 Expression-Based Formalism

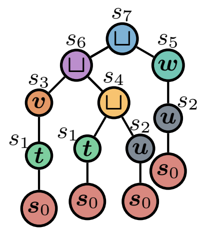
To define invariant confluence formally, we represent a state produced by a system execution as a simple expression generated by the grammar

$$e ::= s \mid t(e) \mid e_1 \sqcup e_2$$

where s represents a state in S and t represents a transaction in T. As an example, consider the system execution in Figure 1a in which a distributed object is replicated across servers p_1, p_2, and p_3. Server p_3 begins with state s_0, transitions to state s_2 by executing transaction u, transitions to state s_5 by executing transaction w, and then transitions to state s_7 by merging with server p_1. Similarly, server p_1 ends up with state s_6 after executing transactions t and v and merging with server p_2. In Figure 1b, we see the abstract syntax tree of the corresponding expression for state s_7.


p_1 s_0 t s_1 v s_3 s_6
p_2 s_0 u s_2 s_4
p_3 s_0 u s_2 w s_5 s_7

(a) System Execution


s_7 $\sqcup$
s_6 $\sqcup$ s_5 w
s_3 v s_4 $\sqcup$ s_2 u
s_1 t s_1 t s_2 u s_0
s_0 s_0 s_0

(b) Expression

Figure 1: A system execution and corresponding expression

We say an expression e is (s_0, T, I)**-reachable** if it corresponds to a valid execution of our system model. Formally, we define $\text{reachable}_{(s_0,T,I)}(e)$ to be the predicate that satisfies the following conditions:

- $\text{reachable}_{(s_0,T,I)}(s_0)$.
- For all expressions e and for all transactions t in the set T of transactions, if $\text{reachable}_{(s_0,T,I)}(e)$ and $I(t(e))$, then $\text{reachable}_{(s_0,T,I)}(t(e))$.
- For expressions e_1 and e_2, if $\text{reachable}_{(s_0,T,I)}(e_1)$ and $\text{reachable}_{(s_0,T,I)}(e_2)$, then $\text{reachable}_{(s_0,T,I)}(e_1 \sqcup e_2)$.

Similarly, we say a state $s \in S$ is (s_0, T, I)-reachable if there exists an (s_0, T, I)-reachable expression e that evaluates to s. Returning to Example 1 with start state $s_0 = 42$, we see that all integers greater than or equal to 42 (i.e. $\{x \in \mathbb{Z} \mid x \geq 42\}$) are (s_0, T, I)-reachable, and all other integers are (s_0, T, I)-unreachable.

Finally, we say O is **invariant confluent** with respect to s_0, T, and I, abbreviated **(s_0, T, I)-confluent**, if all reachable states satisfy the invariant:

$$\{s \in S \mid \text{reachable}_{(s_0,T,I)}(s)\} \subseteq I$$

3. INVARIANT CLOSURE

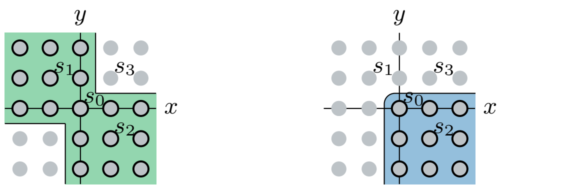
Our ultimate goal is to write a program that can automatically decide whether a given distributed object O is (s_0, T, I)-confluent. Such a program has to automatically prove or disprove that every reachable state satisfies the invariant. However, automatically reasoning about the possibly infinite set of reachable states is challenging, especially because transactions and merge functions can be complex and can be interleaved arbitrarily in an execution. Due to this complexity, existing systems that aim to automatically decide invariant confluence instead focus on deciding a sufficient condition for invariant confluence—dubbed **invariant closure**—that is simpler to reason about [16, 15]. In this section, we define invariant closure and study why the condition is sufficient but not necessary. Armed with this understanding, we present conditions under which it is both sufficient and necessary.

We say an object $O = (S, \sqcup)$ is **invariant closed** with respect to an invariant I, abbreviated **I-closed**, if invariant satisfying states are closed under merge. That is, for every state $s_1, s_2 \in S$, if $I(s_1)$ and $I(s_2)$, then $I(s_1 \sqcup s_2)$.

Theorem 1. *Given an object $O = (S, \sqcup)$, a start state $s_0 \in S$, a set of transactions T, and an invariant I, if $I(s_0)$ and if O is I-closed, then O is (s_0, T, I)-confluent.*

Theorem 1 states that invariant closure is sufficient for invariant confluence. Intuitively, our system model ensures that transaction execution preserves the invariant, so if merging states also preserves the invariant and if our start state satisfies the invariant, then inductively it is impossible for us to reach a state that doesn't satisfy the invariant.

This is good news because checking if an object is invariant closed is more straightforward than checking if it is invariant confluent. Existing systems typically use an SMT solver like Z3 to check if an object is invariant closed [8, 4, 10]. If it is, then by Theorem 1, it is invariant confluent. Unfortunately, invariant closure is *not* necessary for invariant confluence, so if an object is *not* invariant closed, these systems cannot conclude that the object is *not* invariant confluent. The reason why invariant closure is not necessary for invariant confluence is best explained through an example.



(a) Invariant **(b) Reachable points**
Figure 2: An illustration of Example 2

Example 2. Let $O = (\mathbb{Z} \times \mathbb{Z}, \sqcup)$ consist of pairs (x, y) of integers where $(x_1, y_1) \sqcup (x_2, y_2) = (\max(x_1, x_2), \max(y_1, y_2))$. Our start state $s_0 \in \mathbb{Z} \times \mathbb{Z}$ is $(0, 0)$. Our set T of transactions consists of two transactions: $t_{x+1}((x, y)) = (x + 1, y)$ which increments x and $t_{y-1}((x, y)) = (x, y - 1)$ which decrements y. Our invariant $I = \{(x, y) \in \mathbb{Z} \times \mathbb{Z} \mid xy \leq 0\}$ consists of all points (x, y) where the product of x and y is non-positive.

The invariant and the set of reachable states are illustrated in Figure 2 in which we draw each state (x, y) as a point in space. The invariant consists of the second and fourth quadrant, while the reachable states consist only of the fourth quadrant. From this, it is immediate that the reachable states are a subset of the invariant, so O is invariant confluent. However, letting $s_1 = (-1, 1)$ and $s_2 = (1, -1)$, we see that O is not invariant closed. $I(s_1)$ and $I(s_2)$, but letting $s_3 = s_1 \sqcup s_2 = (1, 1)$, we see $\neg I(s_3)$.

In Example 2, s_1 and s_2 witness the fact that O is not invariant closed, but s_1 is *not reachable*. This is not particular to Example 2. In fact, it is fundamentally the reason why invariant closure is not equivalent to invariant confluence. Invariant confluence is, at its core, a property of reachable states, but invariant closure is completely ignorant of reachability. As a result, invariant-satisfying yet unreachable states like s_1 are the key hurdle preventing invariant closure from being equivalent to invariant confluence. This is formalized by Theorem 2.

Theorem 2. *Consider an object $O = (S, \sqcup)$, a start state $s_0 \in S$, a set of transactions T, and an invariant I. If the invariant is a subset of the reachable states (i.e. $I \subseteq \{s \in S \mid \text{reachable}_{(s_0,T,I)}(s)\}$), then*

$$(I(s_0) \text{ and } O \text{ is } I\text{-closed}) \iff O \text{ is } (s_0, T, I)\text{-confluent}$$

The forward direction of Theorem 2 follows immediately from Theorem 1. The backward direction holds because any two invariant satisfying states s_1 and s_2 must be reachable (by assumption), so their join $s_1 \sqcup s_2$ is also reachable. And because O is (s_0, T, I)-confluent, all reachable points, including $s_1 \sqcup s_2$, satisfy the invariant.

4. INTERACTIVE DECISION PROCEDURE

Theorem 2 tells us that if all invariant satisfying points are reachable, then invariant closure and invariant confluence are equivalent. In this section, we present the interactive invariant confluence decision procedure shown in Algorithm 1, that takes advantage of this result.

A user provides Algorithm 1 with an object $O = (S, \sqcup)$, a start state s_0, a set of transactions T, and an invariant I. The user then interacts with Algorithm 1 to iteratively eliminate unreachable states from the invariant. Meanwhile, the algorithm leverages an invariant closure decision procedure to either (a) conclude that O is or is not (s_0, T, I)-confluent

Algorithm 1 Interactive invariant confluence decision procedure

```
// Return if $O$ is $(s_0, T, I)$-confluent.
function IsInvConfluent($O$, $s_0$, $T$, $I$)
    return $I(s_0)$ and Helper($O$, $s_0$, $T$, $I$, $\{s_0\}$, $\emptyset$)

// $R$ is a set of $(s_0, T, I)$-reachable states.
// $NR$ is a set of $(s_0, T, I)$-unreachable states.
// $I(s_0)$ is a precondition.
function Helper($O$, $s_0$, $T$, $I$, $R$, $NR$)
    closed, $s_1$, $s_2$ $\leftarrow$ IsIclosed($O$, $I - NR$)
    if closed then return true
    Augment $R, NR$ with random search and user input
    if $s_1, s_2 \in R$ then return false
    return Helper($O$, $s_0$, $T$, $I$, $R$, $NR$)
```

or (b) provide counterexamples to the user to help them eliminate unreachable states. After all unreachable states have been eliminated from the invariant, Theorem 2 allows us to reduce the problem of invariant confluence directly to the problem of invariant closure, and the algorithm terminates. We now describe Algorithm 1 in detail. An example of how to use Algorithm 1 on Example 2 is given in Figure 3.

IsInvConfluent assumes access to an invariant closure decision procedure IsIclosed(O, I). IsIclosed(O, I) returns a triple (closed, s_1, s_2). closed is a boolean indicating whether O is I-closed. If closed is true, then s_1 and s_2 are null. Otherwise, s_1 and s_2 are a counterexample witnessing the fact that O is not I-closed. That is, $I(s_1)$ and $I(s_2)$, but $\neg I(s_1 \sqcup s_2)$ (e.g., s_1 and s_2 from Example 2). As we mentioned earlier, we can (and do) implement the invariant closure decision procedure using an SMT solver like Z3 [8].

IsInvConfluent first checks that s_0 satisfies the invariant. s_0 is reachable, so if it does not satisfy the invariant, then O is not (s_0, T, I)-confluent and IsInvConfluent returns false. Otherwise, IsInvConfluent calls a helper function Helper that—in addition to O, s_0, T, and I—takes as arguments a set R of (s_0, T, I)-reachable states and a set NR of (s_0, T, I)-unreachable states. Like IsInvConfluent, Helper(O, s_0, T, I, R, NR) returns whether O is (s_0, T, I)-confluent (assuming R and NR are correct). As Algorithm 1 executes, NR is iteratively increased, which removes unreachable states from I until I is a subset of $\{s \in S \mid \text{reachable}_{(s_0,T,I)}(s)\}$.

First, Helper checks to see if O is $(I - NR)$-closed. If IsIclosed determines that O is $(I - NR)$-closed, then by Theorem 1, O is $(s_0, T, I - NR)$-confluent, so

$$\{s \in S \mid \text{reachable}_{(s_0,T,I-NR)}(s)\} \subseteq I - NR \subseteq I$$

Because NR only contains (s_0, T, I)-unreachable states, then the set of (s_0, T, I)-reachable states is equal to set of $(s_0, T, I - NR)$-reachable states which, as we just showed, is a subset of I. Thus, O is (s_0, T, I)-confluent, so Helper returns true.

If IsIclosed determines that O is *not* $(I - NR)$-closed, then we have a counterexample s_1, s_2. We want to determine whether s_1 and s_2 are reachable or unreachable. We can do so in two ways. First, we can randomly generate a set of reachable states and add them to R. If s_1 or s_2 is in R, then they are reachable. Second, we can prompt the user to tell us directly whether the states are reachable or unreachable.

In addition to labelling s_1 and s_2 as reachable or unreachable, the user can also refine I by augmenting R and NR arbitrarily (see Figure 3 for example). In this step, we also make sure that $s_0 \notin NR$ since we know that s_0 is reachable.

After s_1 and s_2 have been labelled as (s_0, T, I)-reachable or not, we continue. If both s_1 and s_2 are (s_0, T, I)-reachable, then so is $s_1 \sqcup s_2$, but $\neg I(s_1 \sqcup s_2)$. Thus, O is not (s_0, T, I)-confluent, so Helper returns false. Otherwise, one of s_1 and s_2 is (s_0, T, I)-unreachable, so we recurse.

Helper recurses only when one of s_1 or s_2 is unreachable, so NR grows after every recursive invocation of Helper. Similarly, R continues to grow as Helper randomly explores the set of reachable states. As the user sees more and more examples of unreachable and reachable states, it often becomes easier and easier for them to recognize patterns that define which states are reachable and which are not. As a result, it becomes easier for a user to augment NR and eliminate a large number of unreachable states from the invariant. See Figure 3, for example. Once NR has been sufficiently augmented to the point that $I - NR$ is a subset of the reachable states, Theorem 2 guarantees that the algorithm will terminate after one more call to IsIclosed.

5. SEGMENTED INVARIANT CONFLUENCE

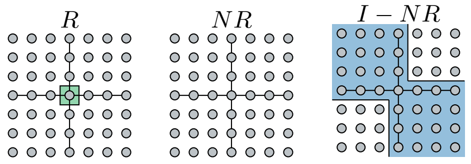
If a distributed object is invariant confluent, then the object can be replicated without the need for any form of coordination to maintain the object's invariant. But what if the object is *not* invariant confluent? In this section, we present a generalization of invariant confluence called **segmented invariant confluence** that can be used to maintain the invariants of non-invariant confluent objects, requiring only a small amount of coordination.

The main idea behind segmented invariant confluence is to segment the state space into a number of segments and restrict the set of allowable transactions within each segment in such a way that the object is invariant confluent *within each segment* (even though it may not be globally invariant confluent). Then, servers can run coordination-free within a segment and need only coordinate when transitioning from one segment to another. We now formalize segmented invariant confluence, describe the system model we use to replicate segmented invariant confluent objects, and introduce a segmented invariant confluence decision procedure.

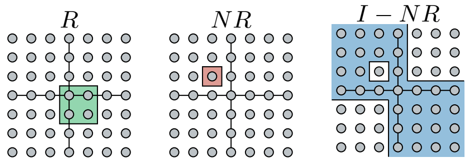
5.1 Formalism

Consider a distributed object $O = (S, \sqcup)$, a start state $s_0 \in S$, a set of transitions T, and an invariant I. A segmentation $\Sigma = (I_1, T_1), \ldots, (I_n, T_n)$ is a sequence of n segments (I_i, T_i) where every T_i is a subset of T and every $I_i \subseteq S$ is an invariant. Note that Σ is a sequence, not a set. The reason for this will become clear in the next subsection. O is **segmented invariant confluent** with respect to s_0, T, I, and Σ, abbreviated (s_0, T, I, Σ)**-confluent**, if the following conditions hold:

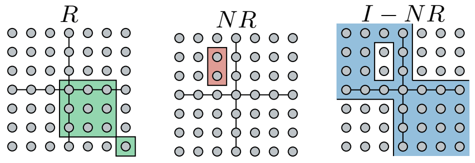
- The start state satisfies the invariant (i.e. $I(s_0)$).
- I is covered by the invariants in Σ (i.e. $I = \cup_{i=1}^{n} I_i$). Note that invariants in Σ do *not* have to be disjoint. That is, they do not have to partition I; they just have to cover I.
- O is invariant confluent within each segment. That is, for every $(I_i, T_i) \in \Sigma$ and for every state $s \in I_i$, O is (s, T_i, I_i)-confluent.



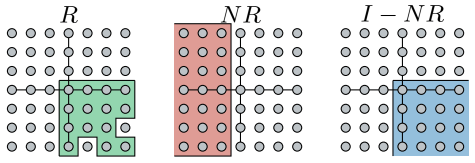
(a) IsInvConfluent determines $I(s_0)$ and then calls Helper with $R = \{s_0\}$, $NR = \emptyset$, and $I = \{(x, y) \mid xy \le 0\}$.



(b) Helper determines that O is not $(I - NR)$-closed with counterexample $s_1 = (-1, 1)$ and $s_2 = (1, -1)$. Helper randomly generates some (s_0, T, I)-reachable points and adds them to R. Luckily for us, $s_2 \in R$, so Helper knows that it is (s_0, T, I)-reachable. Helper is not sure about s_1, so it asks the user. After some thought, the user tells Helper that s_1 is (s_0, T, I)-unreachable, so Helper adds s_1 to NR and then recurses.



(c) Helper determines that O is not $(I - NR)$-closed with counterexample $s_1 = (-1, 2)$ and $s_2 = (3, -3)$. Helper randomly generates some (s_0, T, I)-reachable points and adds them to R. $s_1, s_2 \notin R, NR$, so Helper ask the user to label them. The user puts s_1 in NR and s_2 in R. Then, Helper recurses.



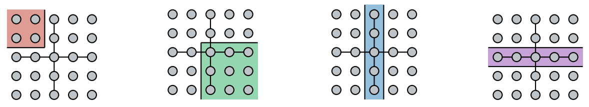
(d) Helper determines that O is not $(I - NR)$-closed with counterexample $s_1 = (-2, 1)$ and $s_2 = (1, -1)$. Helper randomly generates some (s_0, T, I)-reachable points and adds them to R. $s_2 \in R$ but $s_1 \notin R, NR$, so Helper asks the user to label s_1. The user notices a pattern in R and NR and after some thought, concludes that every point with negative x-coordinate is (s_0, T, I)-unreachable. They update NR to $-\mathbb{Z} \times \mathbb{Z}$. Then, Helper recurses. Helper determines that O is $(I - NR)$-closed and returns true!

Figure 3: An example of a user interacting with Algorithm 1 on Example 2. Each step of the visualization shows reachable states R (left), non-reachable states NR (middle), and the refined invariant $I - NR$ (right) as the algorithm executes.

Example 3. Consider again the object $O = (\mathbb{Z} \times \mathbb{Z}, \sqcup)$, transactions $T = \{t_{x+1}, t_{y-1}\}$, and invariant $I = \{(x, y) \mid xy \le 0\}$ from Example 2, but now let the start state s_0 be $(-42, 42)$. O is *not* (s_0, T, I)-confluent because the points $(0, 42)$ and $(42, 0)$ are reachable, and merging these points yields $(42, 42)$ which violates the invariant. However, O is (s_0, T, I, Σ)-confluent for $\Sigma = (I_1, T_1)$, (I_2, T_2), (I_3, T_3), (I_4, T_4) where

$$\begin{aligned} I_1 &= \{(x, y) \mid x < 0, y > 0\} & T_1 &= \{t_{x+1}, t_{y-1}\} \\ I_2 &= \{(x, y) \mid x \ge 0, y \le 0\} & T_2 &= \{t_{x+1}, t_{y-1}\} \\ I_3 &= \{(x, y) \mid x = 0\} & T_3 &= \{t_{y-1}\} \\ I_4 &= \{(x, y) \mid y = 0\} & T_4 &= \{t_{x+1}\} \end{aligned}$$

Σ is illustrated in Figure 4. Clearly, s_0 satisfies the invariant, and I_1, I_2, I_3, I_4 cover I. Moreover, for every $(I_i, T_i) \in \Sigma$, we see that O is I_i-closed, so O is (s, T_i, I_I)-confluent for every $s \in I_i$. Thus, O is (s_0, T, I, Σ)-confluent.



(a) (I_1, T_1). (b) (I_2, T_2). (c) (I_3, T_3). (d) (I_4, T_4).
Figure 4: An illustration of Example 3

5.2 System Model

Now, we describe the system model used to replicate a segmented invariant confluent object without any coordination within a segment and with only a small amount of coordination when transitioning between segments. As before, we replicate an object O across a set $p_1, \ldots, p_n$ of n servers each of which manages a replica $s_i \in S$ of the object. Every server begins with s_0, T, I, and Σ. Moreover, at any given point in time, a server designates one of the segments in Σ as its **active segment**. Initially, every server chooses the first segment $(I_i, T_i) \in \Sigma$ such that $I_i(s_0)$ to be its active segment. We'll see momentarily the significance of the active segment.

As before, servers repeatedly perform one of two actions: execute a transaction or merge states with another server. Merging states in the segmented invariant confluence system model is identical to merging states in the invariant confluence system model. A server p_i sends its state s_i to another server p_j which *must* merge s_i into its state s_j. Transaction execution in the new system model, on the other hand, is more involved. Consider a server s_i with active segment (I_i, T_i). A client can request that p_i execute a transaction t. We consider what happens when $t \in T_i$ and when $t \notin T_i$.

If $t \notin T_i$, then p_i initiates a round of global coordination to execute t. During global coordination, every server temporarily stops processing transactions and transitions to state $s = s_1 \sqcup \ldots \sqcup s_n$, the join of every server's state. Then, every server speculatively executes t transitioning to state $t(s)$. If $t(s)$ violates the invariant (i.e. $\neg I(t(s))$), then every server aborts t and reverts to state s. Then, p_i replies to the client. If $t(s)$ satisfies the invariant (i.e. $I(t(s))$), then every server commits t and remains in state $t(s)$. Every server then chooses the first segment $(I_i, T_i) \in \Sigma$ such that $I_i(t(s))$ to be the new active segment. Note that such a segment is guaranteed to exist because the segment invariants cover I. Moreover, Σ is ordered, as described in the previous subsection, so every server will deterministically pick the same active segment. In fact, an invariant of the system model is that at any given point of normal processing, every server has the same active segment.

Otherwise, if $t \in T_i$, then p_i executes t immediately and without coordination. If $t(s_i)$ satisfies the *active* invariant (i.e. $I_i(t(s_i)))$, then p_i commits t, stays in state $t(s_i)$, and replies to the client. If $t(s_i)$ violates the *global* invariant (i.e. $\neg I(t(s_i)))$, then p_i aborts t, reverts to state s_i, and replies to the client. If $t(s_i)$ satisfies the global invariant but violates the active invariant (i.e. $I(t(s_i))$ but $\neg I_i(t(s_i)))$, then p_i reverts to state s_i and initiates a round of global coordination to execute t, as described in the previous paragraph.

This system model guarantees that all replicas of a segmented invariant confluent object always satisfy the invariant. All servers begin in the same initial state and with the same active segment. Thus, because O is invariant confluent with respect to every segment, servers can execute transactions within the active segment without any coordination and guarantee that the invariant is never violated. Any operation that would violate the assumptions of the invariant confluence system model (e.g. executing a transaction that's not permitted in the active segment or executing a permitted transaction that leads to a state outside the active segment) triggers a global coordination. Globally coordinated transactions are only executed if they maintain the invariant. Moreover, if a globally coordinated transaction leads to another segment, the coordination ensures that all servers begin in the same start state and with the same active segment, reestablishing the assumptions of the invariant confluence system model.

5.3 Interactive Decision Procedure

In order for us to determine whether or not an object O is (s_0, T, I, Σ)-confluent, we have to determine whether or not O is invariant confluent within each segment $(I_i, T_i) \in \Sigma$. That is, we have determine if O is (s, T_i, I_i)-confluent confluent for every state $s \in I_i$. Ideally, we could leverage Algorithm 1, invoking it once per segment. Unfortunately, Algorithm 1 can only be used to determine if O is (s, T_i, I_i)-confluent for a *particular* state $s \in I_i$, not for *every* state $s \in I_i$. Thus, we would have to invoke Algorithm 1 $|I_i|$ times for every segment (I_i, T_i), which is clearly infeasible given that I_i can be large or even infinite.

Instead, we develop a new decision procedure that can be used to determine if an object is (s, T, I)-confluent for every state $s \in I$. To do so, we need to extend the notion of reachability to a notion of coreachability and then generalize Theorem 2. Two states $s_1, s_2 \in I$ are **coreachable** with respect to a set of transactions T and an invariant I, abbreviated **(T, I)-coreachable**, if there exists some state $s_0 \in I$ such that s_1 and s_2 are both (s_0, T, I)-reachable.

Theorem 3. *Consider an object $O = (S, \sqcup)$, a set of transactions T, and an invariant I. If every pair of states in the invariant are (T, I)-coreachable, then*

$$O \text{ is } I\text{-closed} \iff O \text{ is } (s, T, I)\text{-confluent for every } s \in I$$

The proof of the forward direction is exactly the same as the proof of Theorem 1. Transactions always maintain the invariant, so if merge does as well, then every reachable state must satisfy the invariant. For the reverse direction, consider two arbitrary states $s_1, s_2 \in I$. The two points are (T, I)-coreachable, so there exists some state s_0 from which they can be reached. O is (s_0, T, I)-confluent and $s_1 \sqcup s_2$ is (s_0, T, I)-reachable, so it satisfies the invariant.

Algorithm 2 Interactive invariant confluence decision procedure for arbitrary start state $s \in I$

```
// Return if $O$ is $(s, T, I)$-confluent for every $s \in I$.
function IsInvConfluent($O$, $T$, $I$)
    return Helper($O$, $T$, $I$, $\emptyset$, $\emptyset$)

// $R$ is a set of $(T, I)$-coreachable states.
// $NR$ is a set of $(T, I)$-counreachable states.
function Helper($O$, $T$, $I$, $R$, $NR$)
    closed, $s_1$, $s_2$ ← IsIclosed($O$, $I$, $NR$)
    if closed then return true
    Augment $R, NR$ with random search and user input
    if $(s_1, s_2) \in R$ then return false
    return Helper($O$, $T$, $I$, $R$, $NR$)
```

Using Theorem 3, we develop Algorithm 2: a natural generalization of Algorithm 1. Algorithm 1 iteratively refines the set of *reachable* states whereas Algorithm 2 iteratively refines the set of *coreachable* states, but otherwise, the core of the two algorithms is the same. Now, a segmented invariant confluence decision procedure, can simply invoke Algorithm 2 once on each segment.

Example 4. Let $O = (\mathbb{Z}^3 \times \mathbb{Z}^3, \sqcup)$ be an object that separately keeps positive and negative integer counts (dubbed a PN-Counter [23]), replicated on three machines. Every state $s = (p_1, p_2, p_3), (n_1, n_2, n_3)$ represents the integer $(p_1 + p_2 + p_3) - (n_1 + n_2 + n_3)$. To increment or decrement the counter, the ith server increments p_i or n_i respectively, and $\sqcup$ computes an element-wise maximum. Our start state $s_0 = (0, 0, 0), (0, 0, 0)$; our set T of transactions consists of increment and decrement; and our invariant I is that the value of s is non-negative.

Applying Algorithm 1, IsIclosed returns false with the states $s_1 = (1, 0, 0), (0, 1, 0)$ and $s_2 = (1, 0, 0), (0, 0, 1)$. Both are reachable, so O is not (s_0, T, I)-confluent and Algorithm 1 returns false. The culprit is concurrent decrements, which we can forbid in a simple one-segment segmentation $\Sigma = (I, T^+)$ where T^+ consists only of increment transactions. Applying Algorithm 2, IsIclosed again returns false with the same states s_1 and s_2. This time, however, the user recognizes that the two states are not (T^+, I)-coreachable. The user refines NR with the observation that two states are coreachable if and only if they have the same values of n_1, n_2, n_3. After this, IsIclosed (and thus Helper) returns true, and Algorithm 2 terminates.

6. EVALUATION

In this section, we describe and evaluate Lucy: a prototype implementation of our decision procedures and system models. A more complete evaluation can be found in [25]. Lucy includes a Python implementation of the interactive decision procedures described in Algorithm 1 and Algorithm 2. Users specify objects, transactions, invariants, and segmentations in Python. Lucy also includes a C++

implementation of the invariant confluence and segmented invariant confluence system models.

We now evaluate the practicality and efficiency of our decision procedure prototypes. Specifically, we show that specifying objects is not too onerous and that our decision procedures' latencies are small enough to be used comfortably in an interactive way [18].

Example 5 (Foreign Keys)**.** A 2P-Set $X = (A_X, R_X)$ is a set CRDT composed of a set of additions A_X and a set of removals R_X [23]. We view the state of the set X as the difference $A_X - R_X$ of the addition and removal sets. To add an element x to the set, we add x to A_X. Similarly, to remove x from the set, we add it to R_X. The merge of two 2P-sets is a pairwise union (i.e. $(A_X, R_X) \sqcup (A_Y, R_Y) = (A_X \cup A_Y, R_X \cup R_Y)$).

We can use 2P-sets to model a simple relational database with foreign key constraints. Let object $O = (X, Y) = ((A_X, R_X), (A_Y, R_Y))$ consist of a pair of two 2P-Sets X and Y, which we view as relations. Our invariant $X \subseteq Y$ (i.e. $(A_X - R_X) \subseteq (A_Y - R_Y)$) models a foreign key constraint from X to Y. We ran our decision procedure on the object with initial state $((\emptyset, \emptyset), (\emptyset, \emptyset))$ and with transactions that allow arbitrary insertions and deletions into X and Y. After less than a tenth of a second, the decision procedure produced a reachable counterexample witnessing the fact that the object is not invariant confluent. A concurrent insertion into X and deletion from Y can lead to a state that violates the invariant. This object is not invariant confluent and therefore not invariant closed. Thus, existing systems that depend on invariant closure as a sufficient condition are unable to conclude definitively that the object is *not* invariant confluent.

We also reran the decision procedure, but this time with insertions into X and deletions from Y disallowed. In less than a tenth of a second, the decision procedure correctly deduced that the object is now invariant confluent. These results were manually proven in [3], but our tool was able to confirm them automatically in a negligible amount of time.

Example 6 (Escrow Transactions)**.** Escrow transactions are a concurrency control technique that allows a database to execute transactions that increment and decrement numeric values with more concurrency than is otherwise possible with general-purpose techniques like two-phase locking [21]. The main idea is that a portion of the numeric value is put in escrow, after which a transaction can freely decrement the value so long as it is not decremented by more than the amount that has been escrowed. We show how segmented invariant confluence can be used to implement escrow transactions.

Consider again the PN-Counter $s = (p_1, p_2, p_3), (n_1, n_2, n_3)$ from Example 4 replicated on three servers with transactions to increment and decrement the PN-Counter. In Example 4, we found that concurrent decrements violate invariant confluence which led us to a segmentation which prohibited concurrent decrements. We now propose a new segmentation with escrow amount k that will allow us to perform concurrent decrements that respect the escrowed value. The first segment $(\{(p_1, p_2, p_3), (n_1, n_2, n_3) \mid p_1, p_2, p_3 \geq k \wedge n_1, n_2, n_3 \leq k\}, T)$ allows for concurrent increments and decrements so long as every $p_i \geq k$ and every $n_i \leq k$. Intuitively, this segment represents the situation in which every server has escrowed a value of k. Each server can decrement freely, so long as they don't exceed their escrow budget of k. The second segment is the one presented in Example 4 which prohibits concurrent decrements. We ran our decision procedure on this example and it concluded that it was segmented invariant confluent in less than a tenth of a second and without any human interaction.

Further Decision Procedure Evaluation. In [25], we also specify workloads involving Example 1, an auction application, and TPC-C. Lucy processes all of these workloads, as well as the workloads described above, in less than half a second, and no workload requires more than 66 lines of Python code to specify. This shows that the user burden of specification is not too high and that our decision procedures are efficient enough for interactive use.

System Model Evaluation. In addition to our decision procedures, we also evaluate the performance of distributed objects deployed with segmented invariant confluence [25]. Namely, we show that segmented invariant confluent replication can achieve an order of magnitude higher throughput compared to linearizable replication, but the throughput improvements decrease as we increase the fraction of transactions that require coordination. For example, with 5% coordinating transactions, segmented invariant confluent replication performs over an order of magnitude better than linearizable replication; with 50%, it performs as well; and with 100%, it performs two times worse.

7. RELATED WORK

RedBlue consistency [16], is a consistency model that sits between causal consistency and linearizability. In [16], Li et al. introduce invariant safety as a sufficient (but not necessary) condition for RedBlue consistent objects to be invariant confluent. Invariant safety is an analog of invariant closure. In [15], Li et al. develop sophisticated techniques for deciding invariant safety that involve calculating weakest preconditions. These techniques are complementary to our work and can be used to improve the invariant closure subroutine used by our decision procedures.

The homeostasis protocol [22], a generalization of the demarcation protocol [6], uses program analysis to avoid unnecessary coordination between servers in a *sharded* database (whereas invariant confluence targets *replicated* databases).

Explicit consistency [5] is a consistency model that combines invariant confluence and causal consistency, similar to RedBlue consistency with invariant safety. Balegas et al. also describe a variety of techniques—like conflict resolution, locking, and escrow transactions [21]—that can be used to replicate workloads that do not meet their sufficient conditions. Segmented invariant confluence is a formalism that can be used to model simple forms of these techniques.

In [10], Gotsman et al. discuss a hybrid token based consistency model that generalizes a family of consistency models including causal consistency, sequential consistency, and RedBlue consistency. The token based approach allows users to specify certain conflicts that are not possible with segmented invariant confluence. However, segmented invariant confluence also introduces the notion of invariant segmentation, which cannot be emulated with the token based approach. For example, it is difficult to emulate escrow transactions with the token based approach.

8. CONCLUSION

This paper revolved around two major contributions. First, we found that invariant closure fails to incorporate a notion of reachability, and using this intuition, we developed conditions under which invariant closure and invariant confluence are equivalent. We implemented this insight in an interactive invariant confluence decision procedure that automatically checks whether an object is invariant confluent, with the assistance of a programmer. Second, we proposed a generalization of invariant confluence, segmented invariant confluence, that can be used to replicate non-invariant confluent objects with a small amount of coordination while still preserving their invariants.

9. ACKNOWLEDGMENTS

A big thanks to Alan Fekete, Peter Alvaro, Alvin Cheung, Alexandra Meliou, Anthony Tan, and Cristina Teodoropol. This research is supported in part by DHS Award HSHQDC-16-3-00083, NSF CISE Expeditions Award CCF-1139158, and gifts from Alibaba, Amazon Web Services, Ant Financial, CapitalOne, Ericsson, GE, Google, Huawei, Intel, IBM, Microsoft, Scotiabank, Splunk and VMware.

10. REFERENCES

[1] D. Abadi. Consistency tradeoffs in modern distributed database system design: Cap is only part of the story. *Computer*, 45(2):37–42, 2012.

[2] M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. W. Hutto. Causal memory: Definitions, implementation, and programming. *Distributed Computing*, 9(1):37–49, 1995.

[3] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Coordination avoidance in database systems. *PVLDB*, 8(3):185–196, 2014.

[4] V. Balegas, S. Duarte, C. Ferreira, R. Rodrigues, N. Preguiça, M. Najafzadeh, and M. Shapiro. Putting consistency back into eventual consistency. In *Proceedings of the Tenth European Conference on Computer Systems*. ACM, 2015.

[5] V. Balegas, S. Duarte, C. Ferreira, R. Rodrigues, N. Preguiça, M. Najafzadeh, and M. Shapiro. Towards fast invariant preservation in geo-replicated systems. *ACM SIGOPS Operating Systems Review*, 49(1):121–125, 2015.

[6] D. Barbará-Millá and H. Garcia-Molina. The demarcation protocol: A technique for maintaining constraints in distributed database systems. *The VLDB Journal*, 3(3):325–353, 1994.

[7] P. A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys (CSUR)*, 13(2):185–221, 1981.

[8] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[9] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2):51–59, 2002.

[10] A. Gotsman, H. Yang, C. Ferreira, M. Najafzadeh, and M. Shapiro. 'cause i'm strong enough: Reasoning about consistency choices in distributed systems. *ACM SIGPLAN Notices*, 51(1):371–384, 2016.

[11] P. W. Grefen and P. M. Apers. Integrity control in relational database systemsâĂŤan overview. *Data & Knowledge Engineering*, 10(2):187–223, 1993.

[12] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[13] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[14] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[15] C. Li, J. Leitão, A. Clement, N. Preguiça, R. Rodrigues, and V. Vafeiadis. Automating the choice of consistency levels in replicated systems. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 281–292, 2014.

[16] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 265–278, 2012.

[17] R. J. Lipton and J. S. Sandberg. Pram: A scalable shared memory. Technical Report TR-180-88, Computer Science Department, Princeton University, August 1988.

[18] Z. Liu and J. Heer. The effects of interactive latency on exploratory visual analysis. *IEEE transactions on visualization and computer graphics*, 20(12):2122–2131, 2014.

[19] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with cops. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 401–416. ACM, 2011.

[20] S. A. Mehdi, C. Littley, N. Crooks, L. Alvisi, N. Bronson, and W. Lloyd. I can't believe it's not causal! scalable causal consistency with no slowdown cascades. In *NSDI*, pages 453–468, 2017.

[21] P. E. O'Neil. The escrow transactional method. *ACM Transactions on Database Systems (TODS)*, 11(4):405–430, 1986.

[22] S. Roy, L. Kot, G. Bender, B. Ding, H. Hojjat, C. Koch, N. Foster, and J. Gehrke. The homeostasis protocol: Avoiding transaction coordination through program analysis. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1311–1326. ACM, 2015.

[23] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. *A comprehensive study of convergent and commutative replicated data types*. PhD thesis, Inria–Centre Paris-Rocquencourt; INRIA, 2011.

[24] W. Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, 2009.

[25] M. Whittaker and J. M. Hellerstein. Interactive checks for coordination avoidance. *Proceedings of the VLDB Endowment*, 12(1):14–27, 2018.

Technical Perspective of Concurrent Prefix Recovery: Performing CPR on a Database

Philip A. Bernstein
Microsoft Research
philbe@microsoft.com

Where do novel database system research results come from? In the 1970's, most systems research papers proposed mechanisms to support abstractions that were being explored for the first time, such as data translation, indexing, query optimization, high performance transactions, distributed databases, heterogeneous databases, and replicated databases. Novelty was easy to come by. These abstractions now form the core of the database systems field.

Since then, the main abstractions of database systems have not changed much. So where do novel solutions come from now? I claim they are driven by six trends, listed below with some recent examples:

1. New hardware mechanisms – multicore, solid-state disks, vector processing, non-volatile RAM, RDMA, GPUs, FPGAs, enclaves.
2. New software mechanisms – log-structured storage, column storage, transactional memory, blockchain, consensus algorithms, distributed hash tables, machine learning.
3. New data models – key-value stores, XML, JSON, graphs.
4. New system platforms – cloud computing, cloud storage, large main memories, cloud-fog-edge, serverless computing.
5. New workloads – stream processing, OLAP, map-reduce, training and serving ML models, graph algorithms over big data, data science, stateful web services.
6. Different system-level goals – scalability, throughput, consistency, latency, fault tolerance, availability, elasticity, cost, extensibility, security, privacy, manageability, robustness.

There is a well-known repertoire of techniques to address these challenges. They include access control, asynchronous operations, batching, caching, checkpointing, compare-and-swap, compression, cost-based optimization, encryption, function shipping, indirection, lazy updates, locking, materialization, multi-versioning (copy-on-write), parallelism, partitioning, pipelining, pre-fetching, replication, speculation, state machines, timeouts, timestamping, transactional queues, triggers, watchdogs, workflow, and those in (2) above. There are many more of course, but probably not hundreds.

Let us use the paper I am introducing as an example. It addresses the problem of checkpoint and recovery for a transactional key-value store—a well-known workload. Its novelty arises from its ability to scale throughput linearly on a large multicore server with negligible increase of latency and from the way it attains this goal.

To appreciate the significance of the paper's novel contributions, let us consider classical solutions to the problem it solves. A simple approach is partitioning, that is, partition the workload so that each core is responsible for reads and writes on a distinct partition of the database. This would ensure there is no interference between the cores. It would enable each core to log its updates independently of the others, thereby ensuring recoverability. However, it would not be robust with respect to changes in the fraction of operations that are directed to each partition. One core could be overloaded while another has spare capacity.

If we relax the partitioning assumption, the problem becomes much harder. For example, updates by different cores may conflict, which creates dependencies between them. Suppose transaction T updates x in one core and a transaction T' in another core reads x's updated value and updates y. Then if a checkpoint includes that updated value of y, it must also include the updated value of x. That is, the checkpointed state needs to be consistent with respect to updates that were acknowledged to users and with respect to each other. In a classical database system, these problems are addressed by locking and logging. However, locking introduces contention among parallel operations, which limits scalability. Logging also introduces contention, such as the need for parallel threads to coordinate their append operations to the shared log, plus it has many other inefficiencies. Although effective techniques have been developed for many of these problems, logging still poses limits to multicore scalability.

This paper's solution circumvents the scalability bottleneck by combining several techniques: a new recovery model called *concurrent prefix recovery*, the use of a 2-version data model, and a state machine.

With concurrent prefix recovery, the *system* defines a commit point, in contrast to standard techniques where clients issue commit. Clients independently synchronize with this commit request, never blocking each other. In the Rest state (i.e., normal operation), each client runs transactions serially on the latest active version, v. When the system issues a commit, it moves to the Prepare state. Each client periodically reads the system's state. After a client moves into the Prepare state, it continues executing normally. However, if its transaction accesses an item already at version $v + 1$, the transaction aborts, the client moves to the next state, called In-Progress, and the transaction re-executes. After all clients have entered Prepare, the system moves its state to In-Progress. At this point, version v of all items are immutable and can be checkpointed without interference from clients. Now, if a transaction wants to update an item x at version v, it creates a new version $v+1$ of x instead of updating version v. (Another client that is still in Prepare state might see this version $v + 1$, leading to an abort as explained above.) If x is already at version $v + 1$, the transaction updates x in place. After all version v items have been checkpointed, the system returns to the Rest state with $v + 1$ as the latest active version.

Voilà. Simple, scalable, and novel. Read on, for details.

Concurrent Prefix Recovery: Performing CPR on a Database

Guna Prasaad[§]
University of Washington
guna@cs.washington.edu

Badrish Chandramouli
Microsoft Research
badrishc@microsoft.com

Donald Kossmann
Microsoft Research
donaldk@microsoft.com

ABSTRACT

This paper proposes a new recovery model based on group commit, called *concurrent prefix recovery* (*CPR*). CPR differs from traditional group commit implementations in two ways: (1) it provides a semantic description of committed operations, of the form "all operations until time t_i from session i"; and (2) it uses asynchronous incremental checkpointing instead of a WAL to implement group commit in a scalable bottleneck-free manner. CPR provides the same consistency as a point-in-time commit, but allows a scalable concurrent implementation. We used CPR to make two systems durable: (1) a custom in-memory transactional database; and (2) FASTER, our state-of-the-art, scalable, larger-than-memory key-value store. Our detailed evaluation of these modified systems shows that CPR is highly scalable and supports concurrent performance reaching hundreds of millions of operations per second on a multi-core machine.

1. INTRODUCTION

The last decade has seen huge interest in building extremely scalable, high-performance multi-threaded data systems – both databases and key-value stores. Main memory databases exploit multicores (up to 1000s of cores [14]) as well as NUMA, SIMD, HTM, and other hardware advances yielding orders-of-magnitude higher performance than traditional databases. In the open-source FASTER research project [1], we have been developing key-value store technologies that push performance even further. FASTER achieves more than 150M ops/sec on one machine for point updates and lookups, while supporting larger-than-memory data and caching the hot working set in memory [5].

Applications using such systems generally require some form of durability for the changes made to application state. Modern systems can handle extremely high update rates in memory but struggle to retain their high performance when durability is desired. Two broad approaches address this requirement for durability today.

WAL with Group Commit. The traditional approach to achieve durability in databases is to use a *write-ahead log* (*WAL*) that records every change to the database. Group commit amortizes the cost of writing the log to disk as large chunks, but update-intensive applications still stress disk write bandwidth. Even without the I/O bottleneck, a WAL introduces overhead – one study [7] found that 30% of CPU cycles are spent in generating log records due to lock contention, excessive context switching, and buffer contention during logging. Recent research has improved the traditional WAL algorithm along dimensions such as buffer allocation [9], by using thread-local REDO logs [15], and optimizing for small I/Os on flash storage [6]. Johnson et al. [8] propose a distributed group commit using Lamport clocks which reduces the concurrency bottleneck but still incurs log writes. Overall, the overheads of WAL continue to affect scalability today.

Checkpoint-Replay. An alternate to WAL, popular in streaming databases, is to take periodic, consistent, point-in-time checkpoints, and use them with input replay for recovery. Cao et. al. [3] propose asynchronous checkpointing algorithms for applications that are frequently physically consistent i.e. the state of the application is transactionally consistent at a physical point in time. Such a consistent state cannot be attained without quiescing the database in most common scenarios. Traditionally, databases obtain a *fuzzy* checkpoint of its state asynchronously and use the WAL to recover a consistent snapshot during recovery. However, as noted earlier, this approach limits throughput due to the WAL bottleneck. VoltDB [10] uses an asynchronous checkpointing technique which takes checkpoints by making every database record "copy-on-write", and this approach is shown to be expensive in update-intensive workloads [5]. CALC [13] obtains asynchronous consistent checkpoints using an atomic commit log (instead of WAL), in which case the atomic log becomes the new bottleneck. To summarize, existing checkpoint-replay based durability solutions are unable to support the ever growing need for scalability.

These alternatives are depicted in Figs. 1(a) and (b). Both WAL and point-in-time checkpoints have scalability issues. To validate this point, we augmented FASTER with a WAL. An in-memory workload that previously achieved more than 150M ops/sec dropped to around 15M ops/sec after the WAL was enabled, even when writing the log to memory. Creating a copy of data on the log for every update is expensive and stresses contention on the log's tail. Further, we built an in-memory transactional database with WAL and point-in-time checkpoints and found both techniques to bottleneck at around 20M single-key txns/sec (see Sec. 6 for details). This huge performance gap has caused many real deployments to forego durability altogether, e.g., by disabling WAL in RocksDB, or by using workarounds such as approximate recovery and quiesce-and-checkpoint [4]. These approaches introduce complexity, latency, quality, and/or performance penalties.

Our Solution

In this paper, we advocate a different approach. We adopt the semantics of group commit, which commits operations as a batch, as our user

[§]Work started during internship at Microsoft Research.

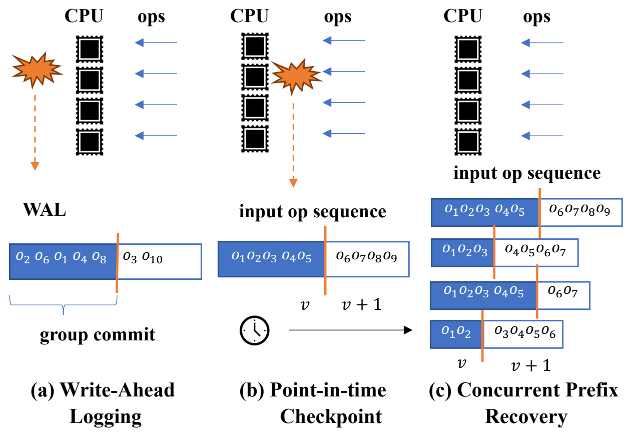

CPU
ops
WAL
$o_2\ o_6\ o_1\ o_4\ o_8$ | $o_3\ o_{10}$
group commit
(a) Write-Ahead Logging
input op sequence
$o_1o_2o_3\ o_4o_5$ | $o_6o_7o_8o_9$
v $v+1$
(b) Point-in-time Checkpoint
input op sequence
$o_1o_2o_3\ o_4o_5$ | $o_6o_7o_8o_9$
$o_1o_2o_3$ | $o_4o_5o_6o_7$
$o_1o_2o_3\ o_4o_5$ | o_6o_7
o_1o_2 | $o_3o_4o_5o_6$
v $v+1$
(c) Concurrent Prefix Recovery

Figure 1: Approaches to Durability

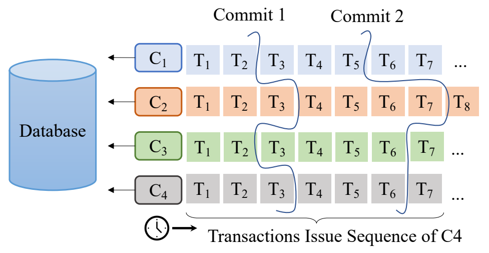

Commit 1
Commit 2
Database
C1 T1 T2 T3 T4 T5 T6 T7 ...
C2 T1 T2 T3 T4 T5 T6 T7 T8
C3 T1 T2 T3 T4 T5 T6 T7 ...
C4 T1 T2 T3 T4 T5 T6 T7 ...
Transactions Issue Sequence of C4

Figure 2: Concurrent Prefix Recovery Model

model for durability. However, instead of acknowledging individual commits, we notify commit as "all operations issued up to time t": we call this model *prefix recovery*. Clients can use this acknowledgment to prune[1] their in-flight operation log until t and expose commit to users. Based on this model, we make the following contributions:

- We argue that it is not possible to guarantee a system-wide prefix recovery without quiescing or introducing a central bottleneck. To address this problem, we propose *concurrent prefix recovery* (*CPR*). In CPR (see Fig. 1(c)), the system periodically notifies each client (or session) S_i of a commit point t_i in its *local* operation timeline, such that all operations before t_i are committed, but none after. We show that CPR has the same consistency as prefix recovery, but allows a scalable asynchronous implementation.
- Traditional group commit is implemented using a WAL. Instead, we implement CPR commits using *asynchronous consistent checkpoints* that capture all changes between commits without introducing any scalability bottleneck. However, this solution requires the ability to take incremental checkpoints very quickly. Fortunately, systems such as FASTER store data in an in-place-updatable log-structured format, making incremental checkpoints very quick to capture and commit. Our approach unifies the worlds of (1) asynchronous incremental checkpoints; and (2) a WAL with group commit, augmented with in-place updates on the WAL between commits.
- While CPR makes it theoretically possible to perform group commit in a scalable asynchronous fashion, it is non-trivial to design systems that achieve these properties without introducing expensive runtime synchronization. To complete the proposal, therefore, we use CPR to build new scalable, non-blocking durability solutions for (1) a custom in-memory transactional database; and (2) FASTER, our state-of-the-art larger-than-memory key-value store. We use an extended version of epoch framework as our building block for loose synchronization, and introduce new state-machine based protocols to perform a CPR commit. As a result, our simple main-memory database implementation scales linearly up to 90M txns/sec – an order-of-magnitude higher than current solutions – while providing periodic CPR commits. Further, our implementation of FASTER with CPR reaches up to 180M ops/sec (the higher throughput compared to [5] is due to a better machine used in this paper) while supporting periodic CPR commits.

To recap, we identify the scalability bottleneck introduced by durability on update-intensive workloads, and propose CPR to alleviate this bottleneck. We then develop solutions to realize CPR in two broad classes of systems: an in-memory database and a larger-than-memory key-value store. Our detailed evaluation shows that it is possible to achieve very high performance in both these CPR-enabled systems, incurring no overhead during normal runtime, and low overhead during commit (in terms of throughput and latency).

[1]Prefix recovery and CPR also work with reliable messaging systems e.g. Kafka, which prunes input messages until some point in time.

2. CONCURRENT PREFIX RECOVERY

A database snapshot is "transactionally consistent" if it reflects all changes made by committed transactions, and none made by uncommitted or in-flight transactions. When the database fails, it can recover to a consistent state using the snapshot, but some in-flight transactions may be lost.

A stricter recovery guarantee is "prefix recovery," where the database – upon failure – can recover to a systemwide prefix of transactions accepted for processing by the database. A naïve method to obtain a prefix recovery snapshot is to stop accepting new transactions until we obtain a consistent snapshot. This technique, called commit-consistent checkpointing [2], forcefully creates a physical point in time at which the database state is consistent, but reduces availability. An alternate method [13] achieves this asynchronously using multiversioning and an atomic commit log. The commit log records every transaction commit and is key to demarcating a prefix that determines which transactions are part of the snapshot. However, the log introduces a scalability bottleneck.

Current state-of-the-art techniques to obtain a prefix recovery snapshot quiesces the database or impedes scalability, neither of which is desirable. We indeed argue that one cannot obtain such a snapshot without these limitations. The key insight is that to obtain the snapshot, we must create a virtual time-point t corresponding to a prefix. As incoming transactions are processed simultaneously, depending on whether they are issued before or after t, they must be executed differently. For example, consider two transactions: T that is accepted before t and T' that is accepted after. Threads must execute T and T' differently as the effect of T must reflect in the snapshot, whereas that of T' should not. So, all threads must agree on a common protocol to determine this unique t, when chosen. To guarantee prefix recovery, threads must coordinate before executing every transaction, which is not possible without introducing a serial communication bottleneck.

To circumvent this fundamental limitation, we introduce CPR. In a prefix recovery snapshot, the database commits all transactions issued before a time-point t. CPR relaxes this requirement by eliminating the need for a "system-wide" time across all clients. Instead, it provides a client-local time, t_C, to each client C, such that all transactions issued by C before t_C are committed and none after t_C are.

Definition 1 (CPR Consistency). *A database state is CPR consistent if and only if, for every client C, the state contains all its transactions committed before a unique client-local time-point t_C, and none after.*

Consider the example shown in Fig. 2. The database has 4 clients issues transactions, each assigned a client-local sequence number. A CPR commit, commit 1 (marked as curve) for instance, commits the transactions $C_1 : \{T_1, T_2\}, C_2 : \{T_1, T_2, T_3\}, C_3 : \{T_1, T_2\}$, and

$C_4 : \{T_1, T_2, T_3\}$. Upon failure, the database recovers the appropriate prefix for each client: for instance, the effects of $\{T_1, T_2, T_3\}$ for client C_2. $C_2 : T_4$ cannot be recovered using commit 1. A later commit, commit 2, persists the effects of transactions until $C_2 : T_7$ including $C_2 : T_4$, and hence $C_2 : T_4$ can then be recovered.

It is desirable to be able to commit the database state at client determined time t_C. For example, concurrent clients issuing update requests as batches of transactions might want to commit at batch boundaries. We claim that client-determined CPR commit cannot be performed without quiescing the database. Let the client-determined set of CPR points for a commit with k clients be $s_1, s_2, ..., s_k$. A transaction request s' by client C_i just after s_i can be executed only when all transactions issued before each of $s_1, s_2, ..., s_k$ have been executed. Hence, s' is blocked till then. Extending this to all clients, the entire database is blocked until all transactions before $s_1, ..., s_k$ have been processed. As a result, client-determined CPR commits are unattainable without blocking. The fundamental limitation here is that s' is blocked because it must read the effects of transactions before CPR points of every client, and these are predetermined (e.g. at a batch boundary). However, in case of CPR, we could circumvent this problem by flipping the roles: clients request for a commit, and the database determines the CPR points for each client *collaboratively* while obtaining the snapshot.

3. EPOCH FRAMEWORK

The epoch framework helps *avoid synchronization between threads whenever possible*. An epoch managed thread executes user operations (e.g. transactions) independently most of the time. It uses thread-local data structures to maintain system state, letting threads lazily synchronize over critical systemwide events. The epoch framework is a key building block in CPR commit protocols. We describe its abstract function here (Refer [5, 12] for details).

We extended the standard epoch framework with custom trigger actions. Threads can register to lazily execute arbitrary global actions, called *trigger actions*, after a global event has occurred. For instance, a thread can register to execute a global action A (e.g. close a file) after a certain thread-local event E happens in every thread (e.g. a thread-local `done` flag set after reading a partition of the file). The key guarantee provided by the framework is that A is executed once and only after all thread-local events have occurred. This functionality is exposed using the following interface:

- `Acquire`: Add the current thread to the epoch managed threads.
- `Refresh`: All epoch managed threads must invoke `Refresh` periodically, but never during an user operation (e.g. only in-between and never in the middle of a transaction).
- `BumpEpoch(cond, action)`: Register ⟨`cond, action`⟩ with the framework; `action` is executed only after `cond` is satisfied.
- `Release`: Remove the current thread from epoch managed threads.

4. CPR COMMIT PROTOCOL

We now present an asynchronous protocol for performing CPR commit in a simple in-memory transactional database. The database has a shared-everything architecture where any thread can access any record. It uses strict 2-Phase Locking with No-Wait deadlock prevention policy for concurrency control. We chose this setup for ease of exposition, and we believe that our algorithm can be easily extended for other protocols as well. We also assume memory twice the size of the database to simplify explanation of the key benefit of CPR.

4.1 Commit Algorithm

Each record in the database has two values, *stable* and *live*, and an integer that stores its current *version*. In steady state, the database is at some version v. A CPR commit corresponds to shifting the database version from v to $(v + 1)$ and capturing its state as of version v. To simplify explanation, we assume a one-to-one mapping between threads and clients: each client C has a dedicated thread T_C to handle all its transactions serially in the order it was issued as shown in Alg. 1. A CPR commit is coordinated using the epoch framework (Sec. 3) as shown in Alg. 2 and its global state machine is shown in Fig. 3.

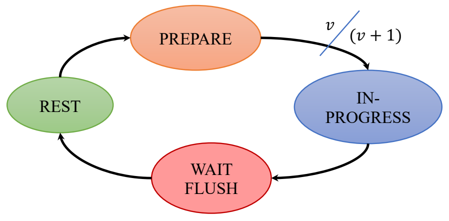


Figure 3: State Machine for CPR Commit in DB

```
Function Run()
    phase, version = Global.phase, Global.version;
    while true do
        repeat
            if inputQueue.TryDequeue(txn) then
                if not Execute(txn, phase, version) then
                    if txn aborted due to CPR then
                        break;
        until k times;
        Refresh();
        newPhase, newVersion = Global.phase, Global.version;
        if phase is PREPARE and newPhase is IN_PROGRESS then
            Record time $t_T$ for thread $T$;
        phase, version = newPhase, newVersion;

Procedure Execute(txn, phase, version)
    foreach (record, accessType) in txn.ReadWriteSet() do
        if record.TryAcquireLock(accessType) then
            lockedRecords.Add(record);
            if phase is PREPARE then
                if record.version > version then
                    Unlock all lockedRecords;
                    Abort txn due to CPR;
            else if phase is IN_PROGRESS or WAIT_FLUSH then
                if record.version < version + 1 then
                    Copy record.live to record.stable;
                    record.version = version + 1;
        else
            Unlock all lockedRecords;
            Abort txn;
    Execute txn using live values;
    Add txn to thread-local staged transactions;
    Unlock all lockedRecords;
```

Algorithm 1: Pseudo-code for Execution Threads

A CPR Commit is lazily coordinated using the epoch framework over three phases: Prepare, In-Progress and Wait-Flush. The protocol state is maintained using two shared global variables, `Global.phase` and `Global.version`. They denote the current phase and version of the database respectively. Threads have a thread-local view of these variables that are updated only during `Refresh`. Avoiding frequent atomic synchronization over these variables is key to the scalability of CPR-based systems and is only possible due to the epoch framework.

Rest Phase. A commit request is issued when the database is in v, Rest. When in Rest, transactions execute normally using strict 2PL with No-Wait policy, the default high-performance phase. The algorithm is triggered by invoking the `Commit` function (Alg. 2). This updates the global state to Prepare and adds an epoch trigger action

```
Function Commit()
    Atomically set Global.phase = PREPARE;
    BumpEpoch(all threads in PREPARE, PrepareToInProg);
Procedure PrepareToInProg()
    Atomically set Global.phase = IN_PROGRESS;
    BumpEpoch(all threads in IN_PROGRESS,
      InProgToWaitFlush);
Procedure InProgToWaitFlush()
    Atomically set Global.phase = WAIT_FLUSH;
    foreach record in database do
        if record.version == Global.version + 1 then
            Capture record.stable;
        else
            Capture record.live;
    Atomically set Global.phase, Global.version = REST,
      Global.version + 1;
    Commit all staged transactions;
```
Algorithm 2: Epoch-based State Machine

`PrepareToInProg`, which is triggered automatically after all threads have entered Prepare. Execution threads update their local view of the phase during subsequent epoch synchronization and enter Prepare.

Prepare Phase. The Prepare phase 'prepares' threads for a CPR Commit. A transaction is executed in Prepare only if all its instructions can be executed on version v of the database. Such transactions are part of the commit and can be recovered on failure. To ensure CPR consistency, they must not read the effects of transactions that are not part of the commit. Upon encountering any record with version greater than v, the transaction immediately aborts, and the thread refreshes its thread-local view of system phase and version. At most one transaction per thread is aborted this way for every commit, since the thread advances to the next phase immediately.

In-Progress Phase. `PrepareToInProg` action is executed automatically after all threads enter Prepare. It updates the system phase to In-Progress and adds another trigger action, `InProgToWaitFlush`. When a thread refreshes its thread-local state now, it enters In-Progress. An In-Progress thread executes transactions in database version $(v+1)$; it updates the version of records it reads/writes to $(v+1)$ when it is $\leq v$. This prevents any transaction belonging to the commit from reading the effects of those that are not. To process $(v+1)$ transactions without blocking, and at the same time capture the record's final value at version v, we copy the live value to the stable value.

Wait-Flush Phase. Once all threads enter In-Progress, the epoch framework executes trigger action `InProgToWaitFlush`. First, it sets the global phase to Wait-Flush, then it captures version v: if a record's version is $(v+1)$, then its stable value is captured, else its live value is captured as part of the commit. Meanwhile, incoming transactions in Wait-Flush are processed similar to those in In-Progress. After all records are captured and persisted, the global phase and version are updated to Rest and $(v+1)$ respectively.

This concludes the CPR commit of version v of the database, resulting in the following theorem (proof sketch in [12]).

Theorem 1 (Correctness). *Algorithms 1 and 2 together produce a transactionally consistent snapshot:*

- *For every thread T, the snapshot reflects all transactions committed before a time t_T, and none after.*
- *The snapshot is conflict-equivalent to a point-in-time snapshot.*

Recovery. Recovery in a CPR-based database is straightforward: we simply load the database back into memory from the latest commit. Unlike traditional WAL-based recovery, there is no need for UNDO processing since the value of each record captured in Alg. 2 is transactionally consistent, and it is the final value after all v transactions have been executed. So, this corresponds to a database state when all

Time	Database State (Before)	Thread 1	Thread 2
1	$A:\langle 1,3,-\rangle, B:\langle 1,2,-\rangle$	🟩 $A=5$	🟩 $B=3$
2	1,Rest $\longrightarrow$ 1,Prepare		
3	$A:\langle 1,5,-\rangle, B:\langle 1,3,-\rangle$	🟩 $B=2$	🟧 $\otimes$
4	$A:\langle 1,5,-\rangle, B:\langle 1,2,-\rangle$	🟧 $\otimes$	🟧 $B=1$
5	1,Prepare $\longrightarrow$ 1,In-Progress		
6	$A:\langle 1,3,-\rangle, B:\langle 1,1,-\rangle$	🟧 $A=5$	🟪 $\otimes$
7	$A:\langle 1,5,-\rangle, B:\langle 1,1,-\rangle$	🟧 $B=7$	🟪 $A=9$
8	$A:\langle 2,9,5\rangle, B:\langle 1,7,-\rangle$	🟪 ~~$A=3$~~ $\Longrightarrow$ $\otimes$	🟪 $B=5$
9	1,In-Progress $\longrightarrow$ 1,Wait-Flush		
10	$A:\langle 2,9,5\rangle, B:\langle 2,5,7\rangle$	🟥 $\otimes$	🟪 $A=3$
11	$A:\langle 2,3,5\rangle, B:\langle 2,5,7\rangle$	🟥 $A=9$	🟥 $\otimes$
12	1,Wait-Flush $\longrightarrow$ 2,Rest		
13	$A:\langle 2,9,5\rangle, B:\langle 2,5,7\rangle$	🟩 $\otimes$	🟥 $A=1$
14	$A:\langle 2,1,5\rangle, B:\langle 2,5,7\rangle$	🟩 $B=4$	🟩 $\otimes$
15	$A:\langle 2,1,5\rangle, B:\langle 2,4,7\rangle$	🟩	🟩

🟩	Rest	🟧	Prepare	🟪	In-Progress
🟥	Wait-Flush	$\otimes$	Epoch-Refresh	key: $\langle$version, live, stable$\rangle$	

Figure 4: Sample Execution of CPR Algorithm

transactions issued before time t_T for every thread T have been committed. Transactions issued after t_T by thread T are lost, as per the definition of CPR-consistency.

4.2 CPR By Example

As an example, we illustrate CPR on two threads for a database that has two records, A and B, see Fig. 4. Each row denotes a time step in which threads execute a 1-key write transaction: for instance $A = 5$ is a transaction that updates A's value to 5. A thread updates its thread-local state during epoch refresh (denoted using $\otimes$). Initially, both threads are in Rest, processing transactions by updating the live values. We receive a commit request at $t = 2$, which updates the global phase to Prepare. Threads 1 and 2 enter Prepare at $t = 4$ and $t = 3$ respectively. Prepare threads also check if record version $>$ current database version (i.e. 1), before executing the transactions.

Since all threads have entered Prepare, the system advances to the In-Progress phase at $t = 5$. Thread 2 enters In-Progress by refreshing its epoch at $t = 6$. This transition from Prepare to In-Progress demarcates its CPR-point. When a record version is 1, In-Progress threads copy its live value to stable value and update the version before processing the transaction. At $t = 7$, thread 2 copies 5, the live value of A, to stable value, updates version to 2 and writes 9 to live value. Thread 1, which is still in Prepare, tries to update A at $t = 8$ but aborts since its version is greater than 1 and immediately refreshes its epoch. Thread 1 enters In-Progress now, marking its CPR-point. As all threads are in In-Progress, the system enters the Wait-Flush phase. We capture the stable values, $A = 5$ and $B = 7$, in the background while threads execute transactions belonging to version 2 on the live values. For other records with version ≤ 1, the live value is captured as part of the commit. Once the captured values are safely persisted on disk, the system transits to Rest with version 2. This ends the CPR commit of version 1 of the database with CPR-points $t = 8$ and $t = 6$.

5. CPR IN FASTER

We next show how CPR-based durability is added to FASTER [5], our recent open-source concurrent latch-free hash key-value store. It supports reads, blind upserts, and read-modify-write (RMW) operations over larger-than-memory data. In the FASTER paper, we report a scalable in-memory throughput of more than 150M ops/sec for the working set in memory, making it a good candidate to apply CPR.

FASTER has two main components, a *hash index* and a *log-structured record store* called `HybridLog`. `HybridLog` defines a *logical address space* that spans secondary storage and main memory. Each record contains some metadata, a key, and a value. Records corresponding to keys that share the same slot in the hash index are organized as a reverse linked list: each record's metadata contains the logical address of the previ-

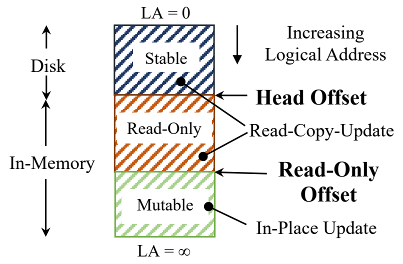


Figure 5: HybridLog Organization in FASTER

ous record mapped to that slot. The hash index points to the logical address of the latest (tail) record in this linked list.

The HybridLog address space (Fig. 5) is divided into an immutable stable region (on disk), an immutable read-only region (in memory), and a mutable region (also in memory). The *head offset* tracks the smallest logical address available in memory. The *read-only offset* divides the in-memory portion of the log into *immutable* and *mutable* regions. The *tail offset* points to the next free address at the tail of the log. FASTER threads perform in-place updates in the hot mutable region for high in-memory performance. Updates to the immutable region use read-copy-update, where a new mutable copy of the record is created at the end of tail to update it. FASTER uses epoch protection to control access to shared memory in a latch-free manner.

5.1 Towards Adding Durability

By default, the index and in-memory portion of HybridLog is lost on failure. We added the ability to periodically commit in-flight operations in the mutable region using CPR, by adding a session-based persistence API to FASTER. Clients can start and end a *session*, identified by a unique Guid, using StartSession and StopSession. Every operation such as Upsert on FASTER occurs within a session, and carries a monotonic session-local serial number. On failure, a client can re-establish a session by invoking ContinueSession with its session Guid as parameter. This call returns the last serial number (CPR point) that FASTER has recovered on that session. As described earlier, CPR commits are session-local, and FASTER recovers to a specific CPR point for every session. The client can also register a callback to be notified of new CPR points whenever FASTER commits.

FASTER provides threads unrestricted access to records in the mutable region of HybridLog, letting user code control concurrency. As CPR enforces a strict *only and all* policy, it is challenging to obtain a CPR-consistent checkpoint without compromising on fast concurrent memory access.

5.2 Asynchronous I/O and CPR

FASTER supports disk-resident data using an asynchronous model: an I/O request is issued in the background, while the requesting thread processes future requests. The user-request is executed later once the record is retrieved from disk. FASTER supports two CPR modes. In the *strict* mode, pending operations logically occur at the point they were originally issued. We also support a *relaxed* mode, where pending operations are re-ordered to logically occur at the time of continuation after I/O completion.

Asynchronous I/O complicates strict CPR in a fundamental way since some requests before a CPR point may be pending. Recall that in CPR, a request r_1 not belonging to the commit must not be executed before a request r_2, potentially from a different session, belonging to the commit. This requirement can lead to quiescing when handled naively; we assume strict CPR and address the issue in our solution.

5.3 HybridLog Checkpoint

We augmented the per-record header in HybridLog to include a version number v for a record. During normal processing, FASTER is in the Rest phase and at a particular version v. HybridLog checkpointing involves (1) shifting the version from v to $(v+1)$; and (2) capturing modifications made during version v. We leverage our epoch framework (Sec. 3) to loosely coordinate a global state machine (see Fig. 6a) for CPR checkpointing without affecting user-space performance. It consists of 5 states: Rest, Prepare, In-Progress, Wait-Pending, and Wait-Flush; state transitions are realized by FASTER threads lazily, when they refresh their epochs. A sample execution with 4 threads is shown in Fig. 6b. Following is a brief overview of each phase:

- Rest: Normal processing on FASTER version v, with identical performance to unmodified FASTER.
- Prepare: Requests accepted before and during the Prepare phase for every thread are part of v commit.
- In-Progress: Transition from Prepare to In-Progress demarcates a CPR point for a thread: requests accepted in In-Progress (or later) phases do not belong to v commit.
- Wait-Pending: Complete pending v requests (in strict CPR only).
- Wait-Flush: Unflushed v records are written to disk asynchronously.
- Rest: Normal processing on FASTER version $(v+1)$.

A CPR commit request (from user or triggered periodically) first records the current tail offset of HybridLog, say L_s^h, and updates the global state from Rest to Prepare. Threads enter Prepare during their subsequent epoch refresh.

Prepare. A Prepare thread T processes an incoming user-request under a shared latch on the key's bucket. When the shared-latch acquisition fails or when the record version is $> v$, T detects that the CPR shift has begun and refreshes its epoch immediately, entering the In-Progress phase. If it never encounters such a scenario, the CPR shift happens during a subsequent epoch refresh. Additionally, in strict CPR, all pending requests are associated with a held shared latch.

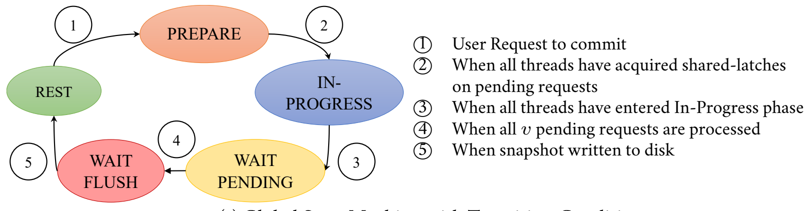
In-Progress. After all threads enter the Prepare phase, the state machine advances to In-Progress. A thread demarcates its CPR point at its transition from Prepare to In-Progress. It now processes requests as belonging to version $(v+1)$. Accessed records in the mutable region are handled carefully. If the record version is $(v+1)$, the thread modifies it in-place as usual. If the record has version $\leq v$, it acquires an exclusive latch on the key's bucket, performs a read-copy-update, creating an updated $(v+1)$ record at the tail, and releases the latch. If exclusive-latch acquisition fails, the request is added to a thread-local pending list corresponding to version $(v+1)$.

Wait-Pending. When all threads enter In-Progress, FASTER enters Wait-Pending in strict CPR, where pending I/Os in version v get completed by all threads, releasing shared latches.

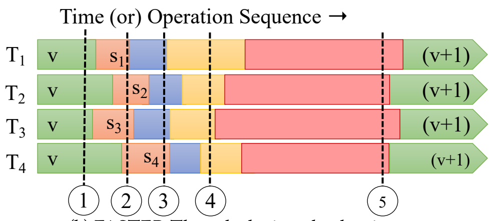
Wait-Flush. Once all v requests have been completed, we record the tail offset of HybridLog, say L_e^h, and shift the read-only offset to L_e^h, which asynchronously flushes HybridLog until L_e^h to disk. Once the asynchronous write to disk is complete, system moves back to Rest with version $(v+1)$. This concludes the HybridLog checkpoint.

5.4 Index Checkpoint

In addition to the HybridLog checkpoint, we obtain a fuzzy checkpoint of the hash index that maps key-hash to logical addresses on HybridLog. The main reason for checkpointing the index is to reduce recovery time by replaying a smaller suffix of the HybridLog during recovery (similar to database checkpoints for WAL truncation). Hence, it can be done much less frequently, particularly with slower log growth due to in-place updates in HybridLog. Since hash bucket entries are updated only using atomic compare-and-swap instructions, the index is always physically consistent. To obtain a fuzzy checkpoint, we write the hash index pages to storage using asynchronous I/O. We also record the tail offset of HybridLog before starting (L_s^i)


1
PREPARE
2
REST
IN-PROGRESS
5
WAIT FLUSH
4
WAIT PENDING
3
① User Request to commit
② When all threads have acquired shared-latches on pending requests
③ When all threads have entered In-Progress phase
④ When all v pending requests are processed
⑤ When snapshot written to disk

(a) Global State Machine with Transition Conditions



(b) FASTER Threads during checkpoint

Figure 6: Overview of CPR for FASTER

and after completion (L^i_e) of the fuzzy checkpoint. We use these offsets during recovery, which is described next.

5.5 Recovery

FASTER recovers to a CPR-consistent state using a combination of a fuzzy hash index and HybridLog checkpoint (say of version v). During recovery, we scan through records in a section of HybridLog, from logical address $S = \min(L^i_s, L^h_s)$ to $E = \max(L^i_e, L^h_e)$, updating the hash index appropriately. The recovered index must point to the latest record with version $\leq v$ for each slot. Due to the fuzzy nature of our index checkpoint, it could point to $(v + 1)$ records or records that are not the latest.

For records in the section of HybridLog between S and E: If the version is $\leq v$, we update the index slot to point to the record's logical address, L_R. When the version is $> v$, we mark the record invalid as it does not belong to v commit of FASTER. Additionally, when the address in the slot is $\geq L_R$, we update the index to point to the previous address stored in the record header. This fix-up may be considered the UNDO phase of our recovery in FASTER. As noted earlier, each slot in the hash index points to a reverse linked-list of records stored in the HybridLog. The copy-on-update scheme in FASTER ensures that records in this list have decreasing logical addresses, while the HybridLog checkpoint design ensures that $(v+1)$ records occur only before all v records in the list. Together, these two invariants result in a consistent FASTER hash index after recovery.

6. EVALUATION

We evaluate CPR in two ways. First, we compare CPR with two state-of-the-art asynchronous durability solutions for a main-memory database: CALC [13] and WAL [11]. Next, we evaluate CPR on our key-value store, FASTER. We present only the key results here and refer the reader to our full paper [12] for a detailed evaluation.

Implementation. For the first part, we implemented a stand-alone main-memory database, that supports three recovery techniques (CPR, CALC, and traditional WAL). Both CALC and CPR implementations have two values, *stable* and *live*, for each record, while WAL only has a single value. An optimal implementation of CPR does not require two values for each record; we do this for a head-to-head comparison with CALC [13]. The entire database is written to disk asynchronously during a CPR/CALC checkpoint. We do not obtain fuzzy checkpoints for WAL but periodically flush the log to disk. All three versions use the main-memory version of FASTER [5] as the data store and implement two-phase locking with NO-WAIT deadlock avoidance policy.

We added CPR to FASTER and that constitutes the second part of our evaluation. Threads first load the key-value store with data, and then issue a sequence of operations. Commit requests are issued periodically. We report system throughput and latency every two seconds. We point FASTER to our SSD, and employ the default expiration based garbage collection scheme (not triggered in these experiments). The total in-memory region of HybridLog is set at 32GB, large enough that reads never hit storage for our workloads, with the mutable region set to 90% of memory at the start. By default, FASTER hash index has #keys/2 hash-bucket entries. We do not directly compare with existing solutions since prior work [5] has shown that other persistent key-value stores such as RocksDB achieve an order of magnitude lower performance (< 1M ops/sec) even when WAL is disabled.

Setup. The first set of experiments are conducted on a Standard D64s v3 machine on Microsoft Azure. The machine has 2 sockets and 16 cores (32 hyperthreads) per socket, 256GB memory and runs Windows Server 2018. Experiments on CPR with FASTER are carried out on a local Dell PowerEdge R730 machine with 2.3GHz Intel Xeon Gold 6140 CPUs, running Windows Server 2016. The machine has 2 sockets and 18 cores (36 hyperthreads) per socket, 512GB memory and a 3.2TB FusionIO NVMe SSD drive. The two-socket experiments shard threads across sockets. We preload input datasets into memory.

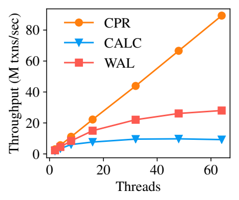
Workloads. For our stand-alone database, we use a mix of transactions based on the Yahoo! Cloud Serving Benchmark (YCSB). Transactions are executed against a single table with 250 million 8 byte keys and 8 byte values. Each transaction is a sequence of read/write requests on these keys, which are drawn from a Zipfian distribution. A request is classified as read or write randomly based on a read-write ratio written as W:R; a read copies the existing value, and a write replaces the value in the database with a provided value. We mainly focus on a low contention ($\theta = 0.1$) workload here since it incurs the most performance penalty due to logging or tail contention.

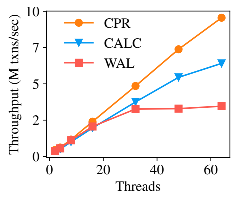
For FASTER with CPR, we use an extended version of the YCSB-A workload, with 250 million distinct 8 byte keys, and value sizes of 8 and 100 bytes. After pre-loading, records occupy 6GB of HybridLog space and the index is 8GB. Workloads are described as R:BU for the fraction of reads and blind updates. We add *read-modify-write* (RMW) updates in addition to the blind updates supported by YCSB. Such updates are denoted as 0:100 RMW in experiments (we only experiment with 100% RMW updates for brevity). RMW updates increment a value by a number from a user-provided input array with 8 entries, to model a per-key "sum" operation. We use the standard Uniform and Zipfian ($\theta = 0.99$) distributions in our workloads.

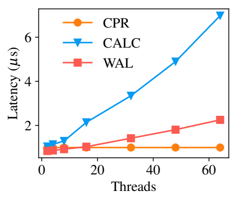
6.1 Evaluation on Transactional Database

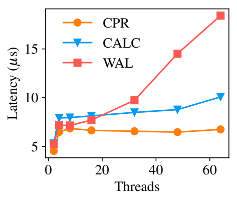
We first plot average throughput (Figs. 7a, 7b) and latency (Figs. 7c, 7d) of the three systems against a varying number of threads for a mixed read-write (50 : 50) workload – for 1- and 10-key transactions. We also profiled the experiment; the breakdown for 1 and 64 threads are shown in Fig. 7e. "Exec" refers to the cost of in-memory transaction processing including acquiring and releasing locks, "Tail-Contention" is the overhead of LSN allocation (in WAL) and appending to the commit log (in CALC), while "Log Write" denotes the cost of writing WAL records on the log.

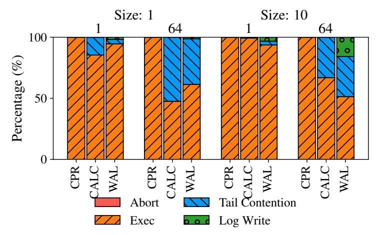
Scalability. CPR scales linearly up to 90M txns/sec on 64 threads for 1-key transactions, whereas CALC and WAL reach a maximum of











(a) Scalability; Size:1 (b) Scalability; Size:10 (c) Latency; Size:1 (d) Latency; Size:10 (e) Analysis

Figure 7: Scalability and Latency on Low Contention (θ= 0.1) YCSB workload

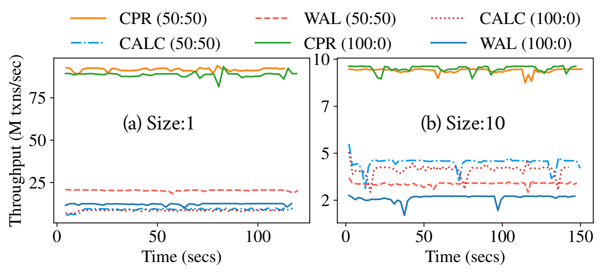


Figure 8: Throughput during Checkpoint

10M txns/sec and 25M txns/sec respectively. The breakdown analysis reveals that tail contention in WAL and in CALC's atomic commit log are a scalability bottleneck. WAL performs better than CALC here since every transaction is appended to the commit log, while 50% read-only 1-key transactions do not generate any WAL records. In case of 10-key transactions, CPR again scales linearly up to 10M txns/secs, while WAL and CALC scale only up to 3.5 and 6.2M txns/sec. Tail contention is still a bottleneck (about $30 - 40\%$) for both CALC and WAL, while WAL incurs an additional 20% overhead for writing log records. Unlike the 1-key case, CALC outperforms WAL since most transactions contain at least one write resulting in a WAL record.

Latency. 1-key transactions (Fig 7c) in CPR are executed in approximately 700 nanoseconds and the latency almost remains constant as we increase the number of threads. This is due to the highly efficient design of the underlying FASTER hash index [5]. Due to tail contention, latency in CALC and WAL increases as we scale. CALC results in a latency of $6\mu s$ on 64 threads, while WAL incurs an average latency of only $2\mu s$ due to 50% read-only transactions. In CPR, 10-key transactions (Fig 7c) incur a cost of $7\mu s$, which is 10x that of a 1-key transaction. CALC latency, even though higher than CPR due to tail contention in the atomic commit log, remains almost constant because the cost of execution is higher in 10-key transactions. Since most 10-key transactions result in a WAL record, the effect of tail contention and writing log records is evident from the increasing trend.

Throughput vs. Time. We now plot average throughput during the lifetime of a run for CPR, CALC and WAL on 64 threads, with checkpoints at 30, 60 and 90 secs both for mixed $(50 : 50)$ and write-only $(100 : 0)$ workloads; Fig. 8a and Fig. 8b correspond to 1- and 10-key transactions respectively. In all three systems, there is no observable drop in throughput during checkpointing. This is due to the asynchronous nature of the solutions. Even for 10-key transactions, the effect of copying over records from live to stable values is minimal as they are already available in upper levels of the cache. CPR design scales better overall and does not involve any serial bottlenecks, yielding a checkpoint throughput of 90M txns/sec. As noted earlier, WAL is better than CALC in $50 : 50$ 1-key transactions due to 50% read-only transactions. The minor difference between write-only and mixed workloads is because writes are more expensive than reads.

6.2 Evaluation of FASTER with CPR

Throughput and Log Size. We plot throughput vs. wall-clock time during the lifetime of a FASTER run. We perform two "full" (index and log) commits during the run, at the 10 sec and 40 sec mark respectively, and plot results for two key distributions (Uniform and Zipf). We evaluate both our commit techniques – *fold-over* and *snapshot* to separate file – in these experiments.

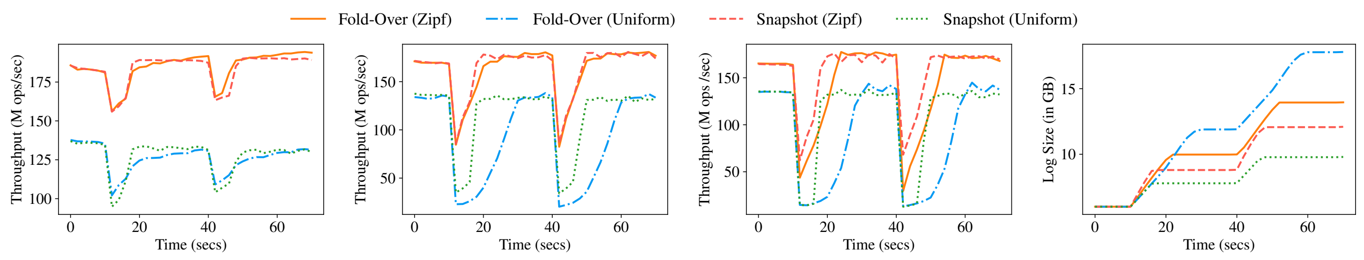
Fig. 9a shows the result for a 90:10 workload (i.e., with 90% reads). Overall, Zipf outperforms Uniform due to better locality of keys in Zipf, reaching up to 180M ops/sec. After commit, both snapshot and fold-over slightly degrade in throughput because of read-copy-updates. It takes 6 secs to write 14GB of index and log, close to the sequential bandwidth of our SSD. After the second commit, the Zipf throughput of fold-over returns to normal faster than snapshot because of its incremental nature. With a 50:50 workload, in Fig. 9b, fold-over drops in throughput after commit, because of the overhead of read-copy-update of records to the tail of `HybridLog`. Performance increases as the working set migrates to the mutable region, with Zipf increasing faster than Uniform as expected. For this workload, snapshot does better than fold-over as it is able to dump the unflushed log to a snapshot file and quickly re-open `HybridLog` for in-place updates. A 0:100 workload with only blind updates demonstrates similar effects, as shown in Fig. 9c. We also profiled execution for the time taken in each CPR phase: each phase lasted for around 5ms, except for Wait-Flush, which took around 6 secs as described above.

Fig. 9d depicts the size of `HybridLog` vs. time, for a 0:100 workload. We note that (1) `HybridLog` size grows slowly with snapshot, as the snapshots are written to a separate file; and (2) `HybridLog` for Uniform grows faster than for Zipf, because more records need to be copied to the tail after a commit for Uniform.

We also experimented with checkpointing only the log, with more frequent commits, since the index is usually checkpointed infrequently. The results are in [12]; briefly, we found CPR commits to have much lower overhead as expected, with a similar trend overall.

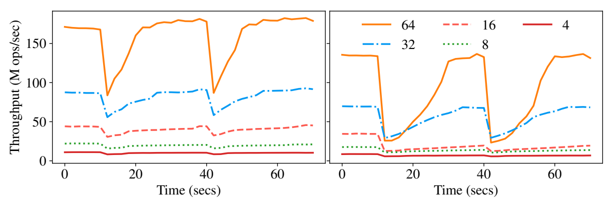
Varying number of threads. We plot throughput vs. time for varying number of threads from 4 to 64, for a 50:50 workload. We depicts the results for Zipf and Uniform distributions in Figs. 10a and 10b respectively, with full fold-over commits taken at the 10 sec and 40 sec mark. Both figures show linear throughput improvement with increasing number of threads, indicating that CPR does not affect scalability. In fact, normal (Rest phase) performance is unaffected by the introduction of CPR. At lower levels of scale, the effect of CPR commits is minimal due to lower Rest phase performance. Further, performance recovery after a commit is faster with more threads, since hot data migrates to mutable region faster.

End-to-End Experiment We evaluate an end-to-end scenario with 36 client threads feeding a 50:50 YCSB workload to FASTER. Each client has a *buffer* of in-flight (uncommitted) requests. When a buffer



(a) Throughput; YCSB 90:10 (b) Throughput; YCSB 50:50 (c) Throughput; YCSB 0:100 (d) Log growth; YCSB 0:100

Figure 9: FASTER Throughput and Log Growth vs. Time; Full Fold-over and Snapshot Commits at $\mathbf{10}$ and $\mathbf{40}$ secs



(a) 50:50 Zipf distribution (b) 50:50 Uniform distribution

Figure 10: Throughput vs. Time; Varying #Threads

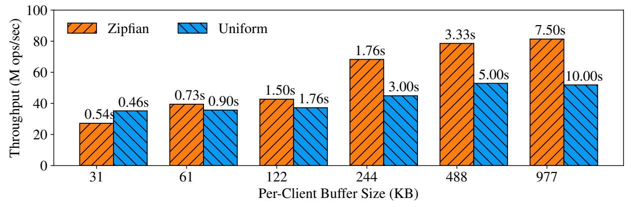


Figure 11: End-to-end Experiment; YCSB $\mathbf{50{:}50}$

reaches 80% capacity, we issue a log-only fold-over commit request, which allows clients to trim their buffers based on CPR points. Clients block if their buffers are full. Each entry in the buffer takes up 16 bytes (for the 8 byte key and value). Fig. 11 shows the results for Zipf and Uniform workloads, as we vary the per-client buffer size. Above each bar is the corresponding average checkpoint interval, or the latency of CPR commit, observed for the given buffer size. We take one full checkpoint, and report average throughput over the next 30 secs.

Increasing the buffer size allows more in-flight operations, which improves throughput for both workloads. Even a small buffer is seen to provide high throughput. For small buffer sizes, commits are issued more frequently (e.g., every 0.5 secs for a 30KB buffer) as expected. The Zipf workload reaches a higher maximum throughput with a larger buffer because the smaller working set reaches the mutable region faster between commits. With the smallest buffer, Uniform outperforms Zipf due to the higher contention faced in Zipf when moving items to the mutable region after every (frequent) commit.

7. CONCLUSION

Modern databases and key-value stores have pushed the limits of multi-core performance to hundreds of millions of operations per second, leading to durability becoming the central bottleneck. Traditional durability solutions have scalability issues that prevent systems from reaching very high performance. We propose a new recovery model based on group commit, called *concurrent prefix recovery* (*CPR*), which is semantically equivalent to a point-in-time commit, but allows a scalable implementation. We present CPR commit protocols for a custom in-memory transactional database and FASTER, our key-value store that supports larger-than-memory data. A detailed evaluation of both systems shows that CPR supports highly concurrent and scalable performance, while providing durability. FASTER with CPR is available as open-source software [1].

8. REFERENCES

[1] FASTER Project. `https://github.com/microsoft/FASTER`.

[2] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.

[3] T. Cao, M. A. V. Salles, B. Sowell, Y. Yue, A. J. Demers, J. Gehrke, and W. M. White. Fast checkpoint recovery algorithms for frequently consistent applications. In *SIGMOD 2011*.

[4] B. Chandramouli, J. Goldstein, M. Barnett, R. DeLine, D. Fisher, J. C. Platt, J. F. Terwilliger, and J. Wernsing. Trill: A high-performance incremental query processor for diverse analytics. *PVLDB*, 8(4):401–412, Dec. 2014.

[5] B. Chandramouli, G. Prasaad, D. Kossmann, J. Levandoski, J. Hunter, and M. Barnett. FASTER: A Concurrent Key-Value Store with In-Place Updates. In *SIGMOD 2018*.

[6] D. Florescu and D. Kossmann. Rethinking cost and performance of database systems. *SIGMOD Record*, 38(1):43–48, 2009.

[7] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker. OLTP Through the Looking Glass, and What We Found There. In *SIGMOD 2008*, SIGMOD '08, pages 981–992, New York, NY, USA, 2008. ACM.

[8] R. Johnson, I. Pandis, R. Stoica, M. Athanassoulis, and A. Ailamaki. Scalability of write-ahead logging on multicore and multisocket hardware. *VLDB J.*, 21(2):239–263, 2012.

[9] H. Jung, H. Han, and S. Kang. Scalable database logging for multicores. *PVLDB*, 11(2):135–148, 2017.

[10] N. Malviya, A. Weisberg, S. Madden, and M. Stonebraker. Rethinking main memory OLTP recovery. In *ICDE 2014*.

[11] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162, Mar. 1992.

[12] G. Prasaad, B. Chandramouli, and D. Kossmann. Concurrent Prefix Recovery: Performing CPR on a Database. In *SIGMOD 2019*.

[13] K. Ren, T. Diamond, D. J. Abadi, and A. Thomson. Low-overhead asynchronous checkpointing in main-memory database systems. In *SIGMOD 2016*.

[14] X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stonebraker. Staring into the abyss: An evaluation of concurrency control with one thousand cores. *PVLDB*, 8(3):209–220, 2014.

[15] W. Zheng, S. Tu, E. Kohler, and B. Liskov. Fast databases with fast durability and recovery through multicore parallelism. In *OSDI 2014*.

Technical Perspective: Constant-Delay Enumeration for Nondeterministic Document Spanners

Benny Kimelfeld
Technion, Israel
bennyk@cs.technion.ac.il

The challenge of extracting structured information from text, or sequential data in general, is prevalent across a multitude of data-science domains. This challenge, known as Information Extraction (IE), instantiates to core components in text analytics, and a plethora of IE paradigms have been developed over the past decades. Rules and rule systems have consistently been key components in such paradigms, yet their roles have varied and evolved over time. Analytics engines such as IBM's SystemT use IE rules for materializing relations inside *relational query languages*. Machine-learning classifiers and probabilistic graphical models (e.g., Conditional Random Fields) use rules for *feature generation*. They also serve as *weak constraints* in Markov Logic Networks (and extensions such as DeepDive), and generators of noisy *training data* in the state-of-the-art Snorkel system.

Originally introduced as the theoretical basis underlying SystemT [5], the framework of *document spanners* provides an abstraction for IE rules [2]. A document spanner states how a document is translated into a relation over its spans. More formally, a *document* is a string **d** over a finite alphabet, a *span* of **d** represents a substring of **d** by its start and end positions, and a *document spanner* is a function that maps every document **d** into a relation over the spans of **d**. The most studied class of document spanners is that of the *regular spanners*—the closure of regular expressions with capture variables under the operators of the Relational Algebra (RA): projection, natural join, union, and difference. Equivalently, the regular spanners are the ones expressible as *Variable-set Automata* (VAs)—nondeterministic finite-state automata that can open and close capture variables.

Past research on document spanners has focused on two main facets: *expressiveness*—which queries can be answered by combining basic text matchers with relational operators? and *complexity*—what is the computational gain of the holistic treatment of the combination, as opposed to the direct way of evaluating relational queries over materialized matchings? The paper "Constant-Delay Enumeration for Nondeterministic Document Spanners" by Amarilli, Bourhis, Mengel and Niewerth [1] makes a substantial leap in our understanding of the second facet.

Prior studies analyzed the complexity of regular spanners under two yardsticks of efficiency. Freydenberger, Kimelfeld, and Peterfreund [4] showed how to compile queries into VAs, and how to evaluate VAs with *polynomial delay in combined complexity*, where both the query and the document are considered input. Florenzano et al. [3] gave algorithms for evaluating VAs in *constant delay in data complexity*, following a linear-time preprocessing phase; this means that, fixing the VA, the evaluation time is (asymptotically) what it takes just to read the document and print the answers one by one. Interestingly, in the first case [4] the delay is inherently dependent on the document size, and in the second case [3] the delay is inherently exponential in the VA size.

While one could suggest that we need to choose between the two yardsticks of efficiency, Amarilli et al. [1] present an algorithm that, surprisingly, delivers both guarantees *at the same time*: in data complexity, their algorithm enumerates with a constant delay following linear-time preprocessing, and in addition, all time intervals are polynomial in the size of the VA. The algorithm is nontrivial, yet quite elegant. In constant-delay algorithms, the crux is typically in the data structure constructed in the preprocessing phase. Here, this data structure is the *mapping DAG* that provides a decision-diagram-like compact representation of the space of answers. In this work, however, a considerable part of the sophistication comes from way that this structure is used at the (constant-delay) enumeration phase. The general idea seems to be useful well beyond the scope of the paper. Importantly, the algorithm has also been implemented and released as open-source in Rust.[1]

1. REFERENCES

[1] A. Amarilli, P. Bourhis, S. Mengel, and M. Niewerth. Constant-delay enumeration for nondeterministic document spanners. In *ICDT*, pages 22:1–22:19, 2019.
[2] R. Fagin, B. Kimelfeld, F. Reiss, and S. Vansummeren. Document spanners: A formal approach to information extraction. *Journal of the ACM*, 62(2):12:1–12:51, 2015.
[3] F. Florenzano, C. Riveros, M. Ugarte, S. Vansummeren, and D. Vrgoc. Constant delay algorithms for regular document spanners. In *PODS*, pages 165–177. ACM, 2018.
[4] D. D. Freydenberger, B. Kimelfeld, and L. Peterfreund. Joining extractions of regular expressions. In *PODS*, pages 137–149. ACM, 2018.
[5] R. Krishnamurthy, Y. Li, S. Raghavan, F. Reiss, S. Vaithyanathan, and H. Zhu. SystemT: A system for declarative information extraction. *SIGMOD Record*, 37(4):7–13, 2008.

[1]`https://github.com/PoDMR/enum-spanner-rs`

Constant-Delay Enumeration for Nondeterministic Document Spanners

Antoine Amarilli
LTCI, Télécom Paris, Institut Polytechnique de Paris
antoine.amarilli@telecom-paris.fr

Pierre Bourhis
CNRS, CRIStAL, UMR 9189 & Inria Lille
pierre.bourhis@univ-lille.fr

Stefan Mengel
CRIL, CNRS & Univ Artois
mengel@cril.fr

Matthias Niewerth
University of Bayreuth
matthias.niewerth@uni-bayreuth.de

ABSTRACT

One of the classical tasks in information extraction is to extract subparts of texts through regular expressions. In the database theory literature, this approach has been generalized and formalized as *document spanners*. In this model, extraction is performed by evaluating a particular kind of automata, called a sequential *variable-set automaton* (VA). The efficiency of this task is then measured in the context of enumeration algorithms: we first run a preprocessing phase computing a compact representation of the answers, and second we produce the results one after the other with a short time between consecutive answers, called the *delay* of the enumeration. Our goal is to have an algorithm that is tractable in combined complexity, i.e., in the sizes of the input document and the VA, while ensuring the best possible data complexity bounds in the input document size, i.e., a constant delay that does not depend on the document. We present such an algorithm for a variant of VAs called *extended sequential VAs* and give an experimental evaluation of this algorithm.

This article is a shortened version of the conference article [4] published at ICDT'19, incorporating experimental results from the journal version [6] currently under review.

1. INTRODUCTION

Information extraction from text documents is an important task in data management. One of the classical approaches is to use regular expressions (*regexes*) with variables to extract subwords satisfying a pattern. For example, to extract the emails addresses in a text, we could extract substrings that contain an @ character, contain no blank character, but are preceded and followed by a blank character. A more general, declarative way to define this task is the framework of *document spanners*, which was first implemented by IBM in their tool SystemT [16], and whose core semantics have then been formalized in [8]. The spanner approach uses variants of regular expressions (namely, *regex formulas* with variables) to extract substrings, and a relational query over these extraction results to combine them. To perform evaluation, the first step is to evaluate the regular expressions, which is done by compiling them to variants of finite automata, the so-called *variable-set automata*, or *VAs* for short. Second, we compute a plan for the relational query, using relational algebra operators like joins, unions and projections. Last, we evaluate this plan over the results of the extraction. The formalization of the spanner framework in [8] has led to a thorough investigation of its properties by the theoretical database community, see [10, 12, 19, 11, 9, 22].

This paper focuses on the first task of efficiently computing the results of the extraction, i.e., computing without duplicates all tuples of ranges of the input document (called *mappings*) that satisfy the conditions described by a VA. As many algebraic operations can in fact be compiled directly into VAs [12], this task actually covers the whole data extraction problem for so-called *regular spanners* [8]. While the extraction task is intractable for general VAs [10], it is known to be tractable if we impose that the VA is *sequential* [12, 9], i.e., if we impose that all accepting runs actually describe a well-formed mapping; we make this assumption throughout our work. Even with this restriction, however, it may still be unreasonable in practice to materialize all mappings: if there are k variables to extract, then mappings are k-tuples and there can be $\Theta(n^{2k})$ mappings on an input document of size n, which is unreasonable if n is large. For this reason, recent works [19, 9, 12] have studied the extraction task in the setting of *enumeration algorithms*: instead of materializing all mappings, we enumerate them one by one while ensuring that the time spent between two consecutive results, called *delay*, is always small. Specifically, [12, Theorem 3.3] has shown how to enumerate

The original version of this paper is entitled "Constant-Delay Enumeration for Nondeterministic Document Spanners" and was published in (22nd International Conference on Database Theory 2019, 2019, Schloss Dagstuhl - Leibniz-Zentrum für Informatik). The authors have been partially supported by the ANR project EQUUS ANR-19-CE48-0019. Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – 431183758.

the mappings with delay linear in the input document and quadratic in the VA, i.e., given a document d and a functional VA A (a subclass of sequential VAs), the delay is $O(|A|^2 \times |d|)$.

Although this result ensures tractability in both the size of the input document and the automaton, the delay may still be as long as $|d|$, which is generally very large. By contrast, enumeration algorithms for other database tasks often enforce stronger tractability guarantees in data complexity [23, 26], in particular *linear preprocessing* and *constant delay* (when measuring complexity in the RAM model with uniform cost measure [1]). Such algorithms consist of two phases: a *preprocessing phase*, which precomputes an index data structure in linear data complexity, and an *enumeration phase*, which produces all results such that the delay between any two consecutive results is always *constant*, i.e., independent from the input data. It was recently shown in [9] that this strong guarantee could be achieved when enumerating the mappings of VAs if we only focus on data complexity, i.e., for any *fixed* VA, we can enumerate its mappings with linear preprocessing and constant delay in the input document. However, the preprocessing and delay in [9] are exponential in the VA because they first determinize it [9, Propositions 4.1 and 4.3]. This is problematic because the VAs constructed from regex formulas [8] are generally nondeterministic and determinization can blow up the size of the automaton exponentially.

Thus, to efficiently enumerate the results of the extraction, we would ideally want to have the best of both worlds: ensure that the *combined complexity* (in the size of the sequential VA and the document) remains polynomial, while ensuring that the *data complexity* (in the document size only) is as small as possible, i.e., linear time for the preprocessing phase and constant time for the delay of the enumeration phase. However, up to now, there was no known algorithm that satisfies both these requirements while working on nondeterministic sequential VAs. Further, it was conjectured that such an algorithm is unlikely to exist [9] because the related task of *counting* the number of mappings is SpanL-hard and thus intractable for such VAs.

The question of nondeterminism is also unsolved for the related problem of enumerating the results of monadic second-order (MSO) queries on words and trees: there are several approaches for this task where the query is given as an automaton, but they require the automaton to be deterministic [7, 2] or their delay is not constant in the input document [18].

Contributions. We show that nondeterminism is in fact not an obstacle to enumerating the results of document spanners efficiently: we present an algorithm that enumerates the mappings of a nondeterministic sequential VA in polynomial combined complexity while ensuring linear preprocessing and constant delay in the input document size. This answers the open question of [9], and improves on the bounds of [12].

The existence of such an algorithm is surprising but in hindsight not entirely unexpected: remember that, in formal language theory, when we are given a word and a nondeterministic finite automaton, then we can evaluate the automaton on the word with tractable combined complexity by determinizing the automaton "on the fly", i.e., computing at each position of the word the set of states where the automaton can be. Our algorithm generalizes this intuition, and extends it to the task of enumerating mappings without duplicates. Here, we present it for so-called *extended sequential VAs*, a variant of sequential VAs introduced in [9]. Note that, despite the name, extended VAs are actually more restrictive than VAs: they can be converted in PTIME to VAs, but the converse is not true as there are VAs for which the smallest equivalent extended VA has exponential size [9]. This being said, our approach also generalizes from sequential extended VAs to sequential VAs: we do not include this extension in this paper for lack of space, but the result can be found in the original paper [4].

Our overall approach is to construct a kind of product of the input document with the extended VA, similarly to [9]. We then use several tricks to ensure the constant delay bound despite nondeterminism; in particular, we precompute a *jump function* that allows us to quickly skip the parts of the document where no variable can be assigned. The resulting algorithm is rather simple and has no large hidden constants. Note that our enumeration algorithm does not contradict the counting hardness results of [9, Theorem 5.2]: while our algorithm *enumerates* mappings with constant delay and without duplicates, we do not see a way to adapt it to *count* the mappings efficiently. This is similar to the enumeration and counting problems for maximal cliques: we can enumerate maximal cliques with polynomial delay [24], but counting them is #P-hard [25].

We have also implemented our algorithm and present a short experimental evaluation using this implementation. The implementation can be found at `https://github.com/PoDMR/enum-spanner-rs` and is under the BSD 3-clause license.

Paper structure. In Section 2, we formally define spanners, VAs, and the enumeration problem that we want to solve on them. We then describe our main result in Section 3, and prove it in Sections 4 and 5. Last, we present the experimental performance of our algorithm in Section 6 and conclude in Section 7.

2. PRELIMINARIES

Document spanners. A *document* $d = d_0 \cdots d_{n-1}$ is just a word over Σ. A *span* of d is a pair $[i, j\rangle$ with $0 \leq i \leq j \leq |d|$, which represents a substring (contiguous subsequence) of d starting at position i and ending at position $j - 1$. To describe the possible results of an information extraction task, we use a finite set $\mathcal{V}$ of variables, and define a result as a *mapping* from these variables to spans of the input document. Following [9, 19] but in contrast to [8], we do not require mappings to assign all variables: formally, a *mapping* of $\mathcal{V}$ on d is a function μ from some domain $\mathcal{V}' \subseteq \mathcal{V}$ to spans of d. We define a *document spanner* to be a function assigning to every input document d a set of mappings, which denotes the set of results of the extraction task on the document d.

Extended VAs. Document spanners are often represented as *variable-set automata* (or *VAs*). We present our results on a variant of VAs introduced by [9], called sequential *extended VAs*. An extended VA on alphabet Σ and variable set $\mathcal{V}$ is an automaton $\mathcal{A} = (Q, q_0, F, \delta)$ where the transition relation δ consists of *letter transitions* of the form (q, a, q') for $q, q' \in Q$ and $a \in \Sigma$, and of *extended variable transitions* (or *ev-transitions*) of the form (q, M, q') where M is a possibly empty set of variable markers ($x\vdash$ or $\dashv x$, $x \in \mathcal{V}$). Intuitively, on ev-transitions, the automaton reads multiple markers at once. A *configuration* of an extended VA is a pair (q, i) where $q \in Q$ and i is a position of the input document d Formally, a *run* σ of $\mathcal{A}$ on $d = d_0 \cdots d_{n-1}$ is a sequence of configurations where letter transitions and ev-transitions alternate:

$$(q_0, 0) \xrightarrow{M_0} (q'_0, 0) \xrightarrow{d_0} (q_1, 1) \xrightarrow{M_1} (q'_1, 1) \xrightarrow{d_1}$$

$$\cdots \xrightarrow{d_{n-1}} (q_n, n) \xrightarrow{M_n} (q'_n, n)$$

where (q'_i, d_i, q_{i+1}) is a letter transition of $\mathcal{A}$ for all $0 \leq i < n$, and (q_i, M_i, q'_i) is an ev-transition of $\mathcal{A}$ for all $0 \leq i \leq n$ where M_i is the set of variable markers *read* at position i.

An extended VAs is called *sequential* if all its accepting runs are *valid* in the following sense: every variable marker is read at most once, and whenever an open marker $x\vdash$ is read at a position i then the corresponding close marker $\dashv x$ is read at a position i' with $i \leq i'$. From each accepting run of an extended sequential VA, we can then define a mapping where each variable $x \in \mathcal{V}$ is mapped to the span $[i, i'\rangle$ such that $x\vdash$ is read at position i and $\dashv x$ is read at position i'; if these markers are not read then x is not assigned by the mapping (i.e., it is not in the domain $\mathcal{V}'$). Throughout this work, we always assume that extended VAs are *sequential*.

The *document spanner* of the VA $\mathcal{A}$ is then the function that assigns to every document d the set of mappings defined by the accepting runs of $\mathcal{A}$ on d: note that the same mapping can be defined by multiple different runs.

The task studied in this paper is the following: given a sequential extended VA $\mathcal{A}$ and a document d, enumerate *without duplicates* the mappings that are assigned to d by the document spanner of $\mathcal{A}$. The enumeration must write each mapping as a set of pairs (m, i) where m is a variable marker and i is a position of d.

In the rest of the paper, we further assume that all extended VAs are *trimmed* in the sense that for every state q there is a document d and an accepting run of the VA where the state q appears. This condition can be enforced in linear time on any sequential VA: we do a graph traversal to identify the accessible states (the ones that are reachable from the initial state), we do another graph traversal to identify the co-accessible states (the ones from which we can reach a final state), and we remove all states that are not accessible or not co-accessible. We implicitly assume that all sequential VAs have been trimmed, which implies that they cannot contain any cycle of variable transitions.

Last, we assume that the states of our extended VAs are partitioned between *ev-states*, from which only ev-transitions originate (i.e., the q_i above), and *letter-states*, from which only letter transitions originate (i.e., the q'_i above); and we impose that the initial state is an ev-state and the final states are all letter-states. Note that transitions reading the empty set move from an ev-state to a letter-state, like all other ev-transitions. This requirement can be imposed in linear time on any input extended VA; because we allow transitions labeled with the empty set, unlike the definition of [9].

EXAMPLE 2.1. *The top of Figure 1 represents a sequential extended VA $\mathcal{A}_0$ to extract email addresses. To keep the example readable, we simply define them as words (delimited by a space or by the beginning or end of document), which contain one at-sign "@" preceded and followed by a non-empty sequence of non-"@" characters. In the drawing of $\mathcal{A}_0$, the initial state q_0 is at the left, and the states q_{10} and q_{12} are final. The transitions labeled by Σ represent a set of transitions for each letter of Σ, and the same holds for Σ', which we define as $\Sigma' := \Sigma \setminus \{@, \textvisiblespace\}$.*

It is easy to see that, on any input document d, there is one mapping of $\mathcal{A}_0$ on d per email address contained in d, which assigns the markers $x\vdash$ and $\dashv x$ to the beginning and end of the email address, respectively. In particular, $\mathcal{A}_0$ is sequential, because any accepting run is valid. Note that $\mathcal{A}_0$ happens to have the property that each mapping is produced by exactly one accepting run, but our results in this paper do not rely on this property.

Matrix multiplication. The complexity bottleneck for some of our results is the complexity of multiplying two Boolean matrices, which is a long-standing open problem, see e.g. [13] for a recent discussion. When stating our results, we often denote by $2 \leq \omega \leq 3$ an exponent for Boolean matrix multiplication: this is a constant such that the product of two r-by-r Boolean matrices can be computed in time $O(r^\omega)$. The best known upper bound is currently $\omega < 2.3728639$, see [14].

3. ENUMERATION RESULT

Our main result is the following.

THEOREM 3.1. *Let $2 \leq \omega \leq 3$ be an exponent for Boolean matrix multiplication. Let $\mathcal{A}$ be a extended sequential VA with variable set $\mathcal{V}$ and with state set Q, and let d be an input document. We can enumerate the mappings of $\mathcal{A}$ on d with preprocessing time in $O((|Q|^{\omega+1} + |\mathcal{A}|) \times |d|)$ and with delay $O(|\mathcal{V}| \times (|Q|^2 + |\mathcal{A}| \times |\mathcal{V}|^2))$, i.e., linear preprocessing and constant delay in the input document, and polynomial preprocessing and delay in the input VA.*

This result is extended to sequential VAs in [4]. Our result implies analogous results for all spanner formalisms that can be translated to sequential VAs. In particular, spanners are not usually written as automata by users, but instead given in a form of regular expressions called *regex-formulas*, see [8] for exact definitions. As we can translate sequential regex-formulas to sequential VAs in linear time [8, 12, 19], our results imply that we can also evaluate them.

Another direct application of our result is for so-called *regular spanners*, which are unions of conjunctive queries

(UCQs) posed on regex-formulas, i.e., the closure of regex-formulas under union, projection and joins. We again point the reader to [8, 12] for the full definitions. As such UCQs can in fact be evaluated by VAs, our result also implies tractability for such representations, as long as we only perform a bounded number of joins.

4. COMPUTING A MAPPING DAG

To show Theorem 3.1, we reduce the problem of enumerating the mappings captured by an extended sequential VA $\mathcal{A}$ to that of enumerating path labels in a special kind of directed acyclic graph (DAG), called a *mapping DAG*. This DAG is intuitively a variant of the product of $\mathcal{A}$ and of the document d, where we represent simultaneously the position in the document and the corresponding state of $\mathcal{A}$. In the mapping DAG, we no longer care about the labels of letter transitions, so we erase these labels and call these transitions *ϵ-transitions*. As for the ev-transitions, we extend their labels to indicate the position in the document in addition to the variable markers. We first give the general definition of a mapping DAG:

Definition 4.1. *A* mapping DAG *consists of a set V of* vertices*, an* initial vertex $v_0 \in V$*, a* final vertex $v_\mathrm{f} \in V$*, and a set of* edges E *where each edge (s, x, t) has a* source vertex $s \in V$*, a* target vertex $t \in V$*, and a* label x*. There are two kinds of edges:* ϵ-edge*, whose label x is ϵ, and* marker edges*, whose label x is a finite (possibly empty) set of pairs (m, i), where m is a variable marker and i is a position. We require that the graph (V, E) is acyclic. We say that a mapping DAG is* normalized *if every path from the initial vertex to the final vertex starts with a marker edge, ends with an ϵ-edge, and alternates between marker edges and ϵ-edges.*

The mapping *$\mu(\pi)$ of a path π in the mapping DAG is the union of labels of the marker edges of π: we require of any mapping DAG that, for every path π, this union is disjoint. Given a set U of vertices of G, we write $\mathcal{M}(U)$ for the set of mappings of paths from a vertex of U to the final vertex; note that the same mapping may be captured by multiple different paths. The set of mappings* captured *by G is then $\mathcal{M}(G) := \mathcal{M}(\{v_0\})$.*

Intuitively, the ϵ-edges correspond to letter transitions of $\mathcal{A}$ (with the letter being erased, i.e., replaced by ϵ), and marker edges correspond to ev-transitions: their labels are a possibly empty finite set of pairs of a variable marker and position, describing which variables have been assigned during the transition. We now explain how we construct a mapping DAG from $\mathcal{A}$ and from a document d, which we call the *product DAG* of $\mathcal{A}$ and d:

Definition 4.2. *Let $\mathcal{A} = (Q, q_0, F, \delta)$ be a sequential extended VA and let $d = d_0 \cdots d_{n-1}$ be an input document. The* product DAG *of $\mathcal{A}$ and d is the normalized mapping DAG whose vertex set is $Q \times \{0, \ldots, n\} \cup \{v_\mathrm{f}\}$. Its edges are:*

- *For every letter-transition (q, a, q') in δ, for every $0 \leq i < |d|$ such that $d_i = a$, there is an ϵ-edge from (q, i) to $(q', i + 1)$;*
- *For every ev-transition (q, M, q') in δ, for every $0 \leq i \leq |d|$, there is a marker edge from (q, i) to (q', i) labeled with the (possibly empty) set $\{(m, i) \mid m \in M\}$.*
- *For every final state $q \in F$, there is an ϵ-edge from (q, n) to v_f.*

The initial vertex of the product DAG is $(q_0, 0)$ and the final vertex is v_f.

Note that, contrary to [9], we do not contract the ϵ-edges but keep them throughout our algorithm.

Example 4.3. *The mapping DAG for our example sequential extended VA $\mathcal{A}_0$ on the document* `a␣a@b␣b@c` *is shown on Figure 1, with the document being written at the left from top to bottom. The initial vertex of the mapping DAG is $(q_0, 0)$ at the top left and its final vertex is v_f at the bottom. We draw marker edges horizontally, and ϵ-edges diagonally. To simplify the example, we only draw the parts of the mapping DAG that are reachable from the initial vertex. Edges are dashed when they cannot be used to reach the final vertex.*

It is clear that the notion of product DAG is a mapping DAG and captures the mappings that we want to enumerate.

Example 4.4. *The set of mappings captured by the example product DAG on Figure 1 is*

$$\{\{(x\vdash, 3), (\dashv x, 5)\}, \{(x\vdash, 6), (\dashv x, 9)\}\},$$

and this is indeed the set of mappings of the example extended VA $\mathcal{A}_0$ on the example document.

Our task is to enumerate $\mathcal{M}(G)$ *without duplicates*, and this is still non-obvious: because of nondeterminism, the same mapping in the product DAG may be witnessed by exponentially many paths, corresponding to exponentially many runs of the nondeterministic extended VA $\mathcal{A}$. We will present in the next section our algorithm to perform this task on the product DAG G. To do this, we need to preprocess G by *trimming* it, and introduce the notion of *levels* to reason about its structure.

First, we present how to *trim* G. We say that G is *trimmed* if every vertex v is both *accessible* (there is a path from the initial vertex to v) and *co-accessible* (there is a path from v to the final vertex). Given a mapping DAG, we can clearly trim it in linear time by two linear-time graph traversals. Hence, we will always implicitly assume that the mapping DAG is trimmed. If the mapping DAG is empty once trimmed, then there are no mappings to enumerate, so our task is trivial. Hence, we assume in the sequel that the mapping DAG is non-empty after trimming. Further, if $\mathcal{V} = \emptyset$ then the only possible mapping is the empty mapping and we can produce it at that stage, so in the sequel we assume that $\mathcal{V}$ is non-empty.

Example 4.5. *For the mapping DAG of Figure 1, trimming eliminates the non-accessible vertices (which are not depicted) and the non-co-accessible vertices (i.e., those with incoming dashed edges).*

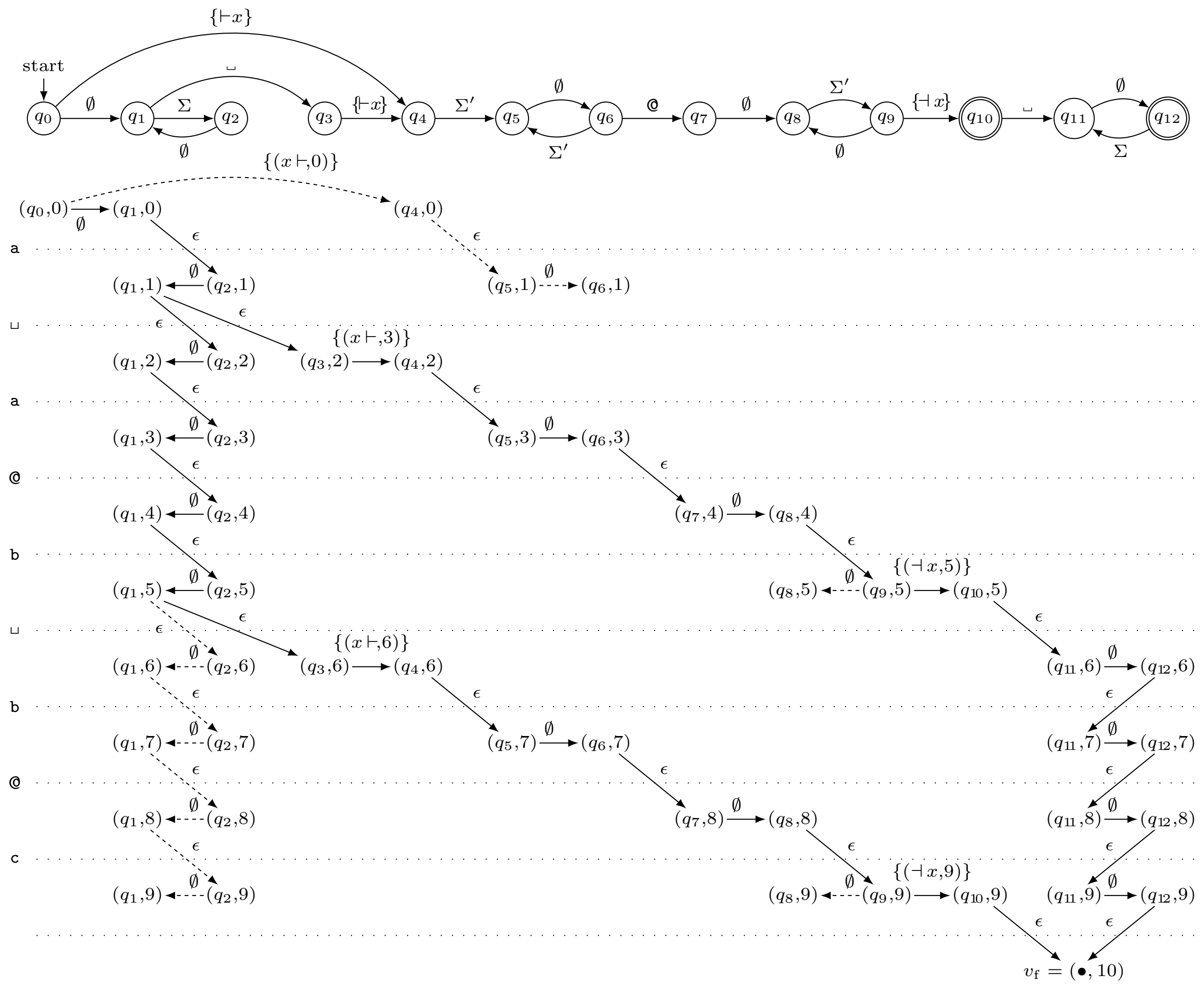


Figure 1: Example sequential extended VA $\mathcal{A}_0$ to extract e-mail addresses (see Example 2.1) and example mapping DAG on an example document (see Examples 4.3, 4.4, 4.5, and 4.7).

Second, we present an invariant on the structure of G by introducing the notion of *levels*:

Definition 4.6. *A mapping DAG G is* leveled *if its vertices $v = (q, i)$ are pairs whose second component i is a nonnegative integer called the* level *of the vertex and written* $\mathsf{level}(v)$*, and where the following conditions hold:*

- *For the initial vertex v_0 (which has no incoming edges), the level is* 0*;*
- *For every ϵ-edge from u to v, it holds that* $\mathsf{level}(v) = \mathsf{level}(u) + 1$*;*
- *For every marker edge from u to v, it holds that* $\mathsf{level}(v) = \mathsf{level}(u)$*. Furthermore, all pairs (m, i) in the label of the edge have $i = \mathsf{level}(v)$.*

The depth *D of G is the maximal level. The* width *W of G is the maximal number of vertices that have the same level.*

The product DAG of $\mathcal{A}$ and d is leveled, W is less than $|Q|$, and D is equal to $|d| + 1$.

Example 4.7. *The example mapping DAG on Figure 1 is leveled, and the levels are represented as horizontal layers separated by dotted lines: the topmost level is level 0 and the bottommost level is level 10.*

In addition to levels, we need the notion of a *level set*:

Definition 4.8. *A* level set *Λ is a non-empty set of vertices in a leveled normalized mapping DAG, that all have the same level (written* $\mathsf{level}(\Lambda)$*) and which are all the source of some marker edge. The singleton $\{v_f\}$ of the final vertex is also considered as a level set.*

In particular, letting v_0 be the initial vertex, the singleton $\{v_0\}$ is a level set. Further, if we consider a level set Λ, which is not the final vertex, then we can follow marker edges from all vertices of Λ (and only such edges) to get to other vertices, and follow ϵ-edges from these vertices (and only such edges) to get to a new level set Λ' with $\mathsf{level}(\Lambda') = \mathsf{level}(\Lambda) + 1$.

5. ENUMERATION ON MAPPING DAGS

In the previous section, we have reduced our enumeration problem for extended VAs on documents to an enumeration problem on normalized leveled mapping DAGs. In this section, we describe our main enumeration algorithm on such DAGs and show the following:

Theorem 5.1. *Let $2 \leq \omega \leq 3$ be an exponent for Boolean matrix multiplication. Given a normalized leveled mapping DAG G of depth D and width W, we can enumerate $\mathcal{M}(G)$ (without duplicates) with preprocessing $O(|G| + D \times W^{\omega+1})$ and delay $O(W^2 \times (r+1))$ where r is the size of each produced mapping.*

Remember that, as part of our preprocessing, we have ensured that the leveled normalized mapping DAG G has been trimmed. We also preprocess G to ensure that, given any vertex, we can access its adjacency list (i.e., the list of its outgoing edges) in some sorted order on the labels, where we assume that $\emptyset$-edges come last. This sorting can be done in linear time on the RAM model [15, Theorem 3.1], so the preprocessing is in $O(|G|)$.

Our general enumeration algorithm is presented as Algorithm 1. We explain the missing pieces next. The function Enum is initially called with $\Lambda = \{v_0\}$, the level set containing only the initial vertex, and with mapping being the empty set.

Algorithm 1 Main enumeration algorithm

```
1: procedure ENUM($G, \Lambda$, mapping)
2:     $\Lambda' :=$ JUMP($\Lambda$)
3:     if $\Lambda'$ is the singleton $\{v_f\}$ of the final vertex then
4:         OUTPUT(mapping)
5:     else
6:         for (locmark, $\Lambda''$) in NEXTLEVEL($\Lambda'$) do
7:             ENUM($G, \Lambda''$, locmark $\cup$ mapping)
```

For simplicity, let us assume for now that the Jump function is just the identity, i.e., $\Lambda' := \Lambda$. As for the call NextLevel(Λ'), it returns the pairs (locmark, Λ'') where:

- The label set locmark is an edge label such that there is a marker edge labeled with locmark that starts at some vertex of Λ'
- The level set Λ'' is formed of all the vertices w at level level(Λ') + 1 that can be reached from such an edge followed by an ϵ-edge. Formally, a vertex w is in Λ'' if and only if there is an edge labeled locmark from some vertex $v \in \Lambda$ to some vertex v', and there is an ϵ-edge from v' to w.

Remember that, as the mapping DAG is normalized, we know that all edges starting at vertices of the level set Λ' are marker edges (several of which may have the same label); and for any target v' of these edges, all edges that leave v' are ϵ-edges whose targets w are at the level level(Λ') + 1.

It is easy to see that the NextLevel function can be computed efficiently:

Proposition 5.2. *Given a leveled trimmed normalized mapping DAG G with width W, and given a level set Λ', we can enumerate without duplicates all the pairs* (locmark, Λ'') $\in$ NextLevel(Λ') *with delay $O(W^2 \times$ |locmark|$)$ in an order such that* locmark $= \emptyset$ *comes last if it is returned.*

The design of Algorithm 1 is justified by the fact that, for any level set Λ', the set $\mathcal{M}(\Lambda')$ can be partitioned based on the value of locmark.

It can easily be proven by induction that Algorithm 1 correctly enumerates $\mathcal{M}(G)$ when Jump is the identity function. However, the algorithm then does not achieve the desired delay bounds: indeed, it may be the case that NextLevel(Λ') only contains locmark $= \emptyset$, and then the recursive call to Enum would not make progress in constructing the mapping, so the delay would not generally be linear in the size of the mapping. To avoid this issue, we use the Jump function to directly "jump" to a place in the mapping DAG where we can read a label different from $\emptyset$. Let us first give the relevant definitions:

Definition 5.3. *Given a level set Λ in a leveled mapping DAG G, the* jump level JL(Λ) *of Λ is the first level $j \geq$* level(Λ) *containing a vertex v' such that some $v \in \Lambda$ has a path to v' and such that v' is either the final vertex or has an outgoing edge with a label which is $\neq \epsilon$ and $\neq \emptyset$. In particular, we have* JL(Λ) = level(Λ) *if some vertex in Λ already has an outgoing edge with such a label, or if Λ is the singleton set containing only the final vertex.*

The jump set *of Λ is then* Jump(Λ) := Λ *if* JL(Λ) = level(Λ), *and otherwise* Jump(Λ) *is formed of all vertices at level* JL(Λ), *to which some $v \in \Lambda$ have a directed path whose last edge is labeled ϵ. This ensures that* Jump(Λ) *is always a level set.*

The definition of Jump ensures that we can jump from Λ to Jump(Λ) when enumerating mappings, and it will not change the result because we only jump over ϵ-edges and $\emptyset$-edges.

What is more, Algorithm 1 now achieves the desired delay bounds, as we will show. Of course, this relies on the fact that the Jump function can be efficiently precomputed and evaluated. We only state this fact here, and give the proof and more details in [4]. Intuitively, the jump function relies on the multiplication of matrices of size $W \times W$, hence the time bound.

Proposition 5.4. *Given a leveled mapping DAG G with width W, we can preprocess G in time $O(D \times W^{\omega+1})$ such that, given any level set Λ of G, we can compute the jump set* Jump(Λ) *of Λ in time $O(W^2)$.*

We can now conclude the proof of Theorem 5.1 by showing that the preprocessing and delay bounds are as claimed. For the preprocessing, this is clear: we do the preprocessing in $O(|G|)$ presented at the beginning of the section (i.e., trimming, and computing the sorted adjacency lists), followed by that of Proposition 5.4. For the delay, we can show that Algorithm 1 has delay $O(W^2 \times (r+1))$, where r is the size of the mapping of each produced path. In particular, the delay is independent of the size of G.

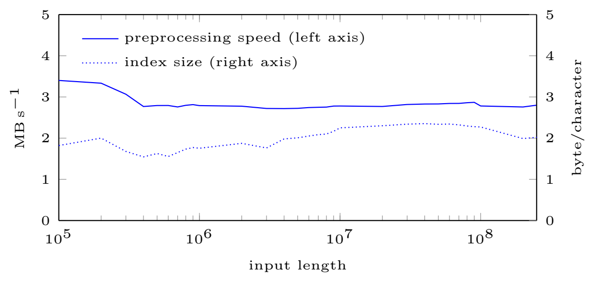


Figure 2: Preprocessing time and index structure size for the query TTAC.{0,1000}CACC on inputs of different lengths.

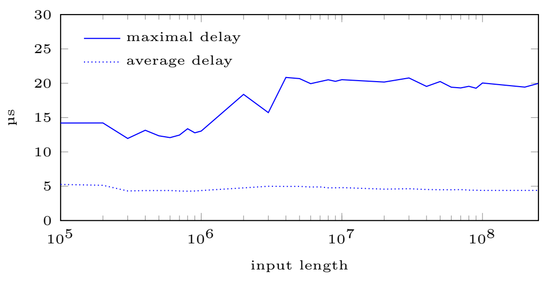


Figure 3: Enumeration delay for the query TTAC.{0,1000}CACC on inputs of different lengths.

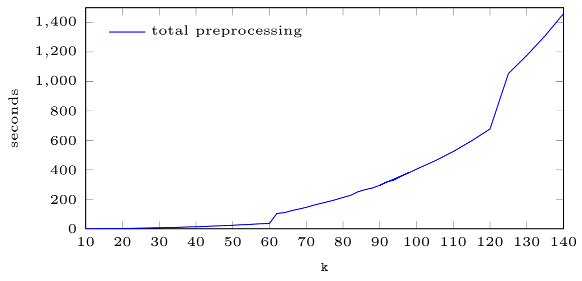


Figure 4: Preprocessing time for the query .{0,k} on an input document of 100 kB, as a function of k.

6. EXPERIMENTS

In this section, we present a very short experimental evaluation of our implementation of the enumeration algorithm. More results can be found in [6]. Our implementation enumerates the mappings assigned to a document by a nondeterministic sequential VA.

The tests were run in a virtual machine that had exclusive access to two Xeon E5-2630 CPU cores. The algorithm is single-threaded, but the additional core was added to minimize the effects of background activity of the operating system.

Measuring the delays between outputs of the algorithm is challenging, because the timescale for these delays is so tiny that unavoidable hardware interrupts can make a big difference. To eliminate outliers resulting from such interrupts, we exploited the fact that our enumeration algorithm is fully deterministic. We ran the algorithm twenty times and recorded all delays. Afterwards, for each produced result, we took the median of the twenty delays that we collected. All delay measurements use this approach, e.g., if we compute the maximum delay for a query, it is actually the maximum over these medians.

We benchmarked our implementation on a genetic dataset: the first chromosome of the human genome reference sequence GRCh38, available at https://www.ncbi.nlm.nih.gov/genome/guide/human/. It contains roughly 250 million base pairs, where each base pair is encoded as a single character. We also use prefixes of this data in the experiments, when we need to benchmark against input documents of various sizes.

We consider the query extracting factors defined by the regex TTAC.{0,100}CACC to illustrate the data complexity of our algorithm, and consider the set of queries extracting all substrings up to a given length k (i.e., the regex .{0,k}) to illustrate its combined complexity.

For the first query, we give in Figure 2 the preprocessing time and size of the index structure divided by the input length, and give in Figure 3 the delay. We see that the preprocessing speed is roughly 3 megabytes per second and the index structure is twice as large as the input document. The average delay is constant (around five microseconds, amounting to 200,000 results per second), while the maximum delay is roughly four times larger.

For the queries of the form .{0,k}, we used as input the first 100,000 characters of the genomic data from the previous experiment. This query does not look interesting and indeed, all resulting mappings can be computed trivially given the length of the string. However, this query triggers the worst case behavior of our algorithm, as almost all levels have width $k+1$. We give the preprocessing time in Figure 4. As our implementation uses the naive $O(n^3)$ matrix multiplication algorithm, its running time is supposed to be $\Theta(k^4)$ in this case. This is consistent with what we observe experimentally. The jumps in preprocessing time that can be seen in the figure result from the fact that our implementation pads the matrix widths to a multiple of 64.

7. CONCLUSION

We have shown that we can efficiently enumerate the mappings of sequential variable-set automata on input documents, achieving linear-time preprocessing and constant delay in data complexity, while ensuring that preprocessing and delay are polynomial in the input VA even if it is not deterministic. This result was previously considered as unlikely by [9], and it improves on the algorithms in [12]: with our algorithm, the delay between outputs does not depend on the input document, whereas it had a linear dependency on the size of the input document in [12].

Since the publication of our original paper [4], we have extended our results in several ways. First, our algorithm has been implemented and we have evaluated its performance experimentally; we summarized these results in Section 6, with the full results being given in [6]. Secondly, we have studied the problem of efficient enumeration on dynamic documents, i.e., maintaining the index structures that we use for enumeration when the input document is updated. Our results in this direction are presented in [5], in the more general setting of enumerating queries over trees. Specifically, relative to [4], we study enumeration for nondeterministic tree automata (rather than word automata), and achieve the same theoretical complexity bounds. Moreover, we can update our index structure in logarithmic time in the size of the tree when performing atomic updates on the input tree, i.e., relabeling a node, deleting or adding a leaf. Our results in [5] thus achieve the same data complexity bounds as the previously proposed algorithms for efficient enumeration of such queries on trees, e.g., those of [3, 18, 17, 20, 21], while supporting a more expressive update language, and while additionally ensuring tractability in the nondeterministic tree automaton.

One remaining open problem for efficient enumeration on dynamic data is to have an efficient support for more general updates. Specifically, in the context of words, our update language from [5] only allows single letter changes in the input documents. We do not know how to deal efficiently with more complex update operators, e.g., bulk update operations that modify large parts of the text at once like cutting and pasting parts of the text, splitting or joining strings, etc. We also do not know how to handle the complexity of updates to avoid the logarithmic dependency in the input document: while we show a lower bound in [5] on the update time, it may be possible to achieve constant-time updates for the case of strings for specific updates, e.g., at the beginning or end of the word, as in the case of rotating a log file, or for more restricted queries than the class of regular spanners. Last, an interesting open question is whether our methods allow for efficient support for other operations, e.g., testing if an input mapping is an answer to the query: such testing queries are efficiently supported in [17] (which has no support for updates), and we do not know if we can handle such queries with our methods (and especially in combination with updates).

8. REFERENCES

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The design and analysis of computer algorithms.* Addison-Wesley, 1974.

[2] A. Amarilli, P. Bourhis, L. Jachiet, and S. Mengel. A circuit-based approach to efficient enumeration. In *ICALP*, 2017.

[3] A. Amarilli, P. Bourhis, and S. Mengel. Enumeration on trees under relabelings. In *ICDT*, 2018.

[4] A. Amarilli, P. Bourhis, S. Mengel, and M. Niewerth. Constant-delay enumeration for nondeterministic document spanners. In *ICDT*, 2019.

[5] A. Amarilli, P. Bourhis, S. Mengel, and M. Niewerth. Enumeration on trees with tractable combined complexity and efficient updates. In *PODS*, 2019.

[6] A. Amarilli, P. Bourhis, S. Mengel, and M. Niewerth. Constant-delay enumeration for nondeterministic document spanners, 2020. `https://arxiv.org/abs/2003.02576`.

[7] G. Bagan. MSO queries on tree decomposable structures are computable with linear delay. In *CSL*, 2006.

[8] R. Fagin, B. Kimelfeld, F. Reiss, and S. Vansummeren. Document spanners: A formal approach to information extraction. *J. ACM*, 62(2), 2015.

[9] F. Florenzano, C. Riveros, M. Ugarte, S. Vansummeren, and D. Vrgoc. Constant delay algorithms for regular document spanners. In *PODS*, 2018.

[10] D. D. Freydenberger. A logic for document spanners. In *ICDT*, 2017.

[11] D. D. Freydenberger and M. Holldack. Document spanners: From expressive power to decision problems. *Theory Comput. Syst.*, 62(4), 2018.

[12] D. D. Freydenberger, B. Kimelfeld, and L. Peterfreund. Joining extractions of regular expressions. In *PODS*, 2018.

[13] F. L. Gall. Improved output-sensitive quantum algorithms for Boolean matrix multiplication. In *SODA*, 2012.

[14] F. L. Gall. Powers of tensors and fast matrix multiplication. In *ISSAC*, 2014.

[15] E. Grandjean. Sorting, linear time and the satisfiability problem. *Annals of Mathematics and Artificial Intelligence*, 16(1), 1996.

[16] IBM Research. SystemT, 2018. `https://researcher.watson.ibm.com/researcher/view_group.php?id=1264`.

[17] W. Kazana and L. Segoufin. Enumeration of monadic second-order queries on trees. *TOCL*, 14(4), 2013.

[18] K. Losemann and W. Martens. MSO queries on trees: Enumerating answers under updates. In *CSL-LICS*, 2014.

[19] F. Maturana, C. Riveros, and D. Vrgoc. Document spanners for extracting incomplete information: Expressiveness and complexity. In *PODS*, 2018.

[20] M. Niewerth. MSO queries on trees: Enumerating answers under updates using forest algebras. In *LICS*, 2018.

[21] M. Niewerth and L. Segoufin. Enumeration of MSO queries on strings with constant delay and logarithmic updates. In *PODS*, 2018.

[22] L. Peterfreund. *The Complexity of Relational Queries over Extractions from Text.* PhD thesis, Technion, 2019. `http://www.cs.technion.ac.il/users/wwwb/cgi-bin/tr-get.cgi/2019/PHD/PHD-2019-10.pdf`.

[23] L. Segoufin. A glimpse on constant delay enumeration (Invited talk). In *STACS*, 2014.

[24] S. Tsukiyama, M. Ide, H. Ariyoshi, and I. Shirakawa. A new algorithm for generating all the maximal independent sets. *SIAM J. Comput.*, 6, 09 1977.

[25] L. Valiant. The complexity of computing the permanent. *Theoretical Computer Science*, 8(2), 1979.

[26] K. Wasa. Enumeration of enumeration algorithms. *CoRR*, 2016.

Technical Perspective: Database Repair Meets Algorithmic Fairness

Lise Getoor
UC Santa Cruz, USA

There has been an explosion of interest in fairness in machine learning. In large part, this has been motivated by societal issues highlighted in a string of well publicized cases such as gender biased job recommendation and racially biased criminal risk prediction algorithms. Both the recognition of the potential disparate impacts of machine learning due to historical bias in the data and the realization of how algorithmic decision making can exaggerate existing structural inequities has become increasingly well known.

This has spawned a growing body of work that examines fairness in ML [1]. From a theoretical perspective, it has opened a pandora's box of new fairness measures, impossibility results, and optimization strategies. However, this line of work has faced criticism. First, the notion that there is any one correct societal fairness definition, and the framing of ML fairness as a simple optimization problem, is suspect. Second, on technical grounds, unless one takes into account the underlying causal structure in the domain, there is no way to untangle, simply from data, whether the data is biased (and hence an algorithm trained on it is fair or not).

The paper "Database Repair Meets Algorithmic Fairness" by Babak Salimi, Bill Howe and Dan Suciu addresses this second criticism directly, and, I would argue, by generalizing the problem setting, they also address the first criticism. Furthermore, they introduce a refreshing database perspective on the problem. The lovely thing about this paper is that it tackles an important real-world issue, offers deep technical contributions, and includes convincing empirical results. Few papers are able to achieve all this, and none that I can think of do it as nicely and concisely.

First, it's useful to review Simpson's paradox, the well-known statistical phenomena that statistical correlations may reverse themselves depending on how data is aggregated. Within the fairness setting, we are interested in whether there is a dependence between a sensitive or (legally) protected attribute (such as gender, race or religion), and a decision outcome (such as admissions, hiring, credit or parole). If that dependence can reverse itself when we condition on another variable, such as age, then making conclusions about fairness will be difficult! Luckily if we have additional information about the underlying causal structure in the domain to reason about confounders we can unpack the correlations between protected attributes and outcomes. Pearl, in a long line of foundational work has developed a calculus of causation that enables one to translate between statistical statements and causal statements in a principled manner [2, 3]. With these tools in hand, when the full causal model is available, it allows us to determine whether there is an inappropriate dependency between a sensitive attribute and a decision.

However, this is a strong requirement. Salimi et al. use Pearl's causal framework as the foundation for a general and flexible construction for introducing admissible and inadmissible attributes while relaxing the requirement of having the full causal model available. Next, the authors draw an elegant connection between causal modeling and database theory to transform the problem of removing bias in data into a database repair problem. They show how to map causal interventions from statistical conditional independence constraints into multi-valued dependencies that should hold in the data. To ensure the statistical independencies required for fairness hold, they generate samples matching the empirical distribution as closely as possible and apply techniques from database repair to modify the data such that independences are satisfied. (Interestingly, this result can be used in *any* setting where one wishes to mix desired interventions and distributional constraints with empirical information.) The authors suggest that the repaired training data "can be seen as a sample from a hypothetical fair world".

To do this, they introduce a notion of justifiable fairness and prove that for a classifier to be justifiably fair, it is sufficient that the outcome variable is conditionally independent of the inadmissible attributes given the admissible attributes. Next, they show how to transform this requirement on a classifier into an integrity constraint on the training data! The paper's contributions include correctly setting up the theoretical machinery to make this translation between probability distributions and databases. While the high level intuition is simple, the details are quite non-trivial.

All in all, this is an important paper, and has something for everyone—real-world impact, theoretical results that bridge causal modeling and database theory, all in an elegant and well-written package.

1. REFERENCES

[1] S. Barocas, M. Hardt, and A. Narayanan. *Fairness and Machine Learning*. 2019. `http://www.fairmlbook.org`.
[2] J. Pearl. *Causality*. Cambridge University Press, 2009.
[3] J. Pearl and D. Mackenzie. *The Book of Why: The New Science of Cause and Effect*. Basic Books, 2018.

Database Repair Meets Algorithmic Fairness

Babak Salimi, Bill Howe, Dan Suciu
University of Washington

ABSTRACT

Fairness is increasingly recognized as a critical component of machine learning systems. However, it is the underlying data on which these systems are trained that often reflect discrimination, suggesting a database repair problem. Existing treatments of fairness rely on statistical correlations that can be fooled by anomalies, such as Simpson's paradox. Proposals for causality-based definitions of fairness can correctly model some of these situations, but they rely on background knowledge of the underlying causal models. In this paper, we formalize the situation as a database repair problem, proving sufficient conditions for fair classifiers in terms of admissible variables as opposed to a complete causal model. We show that these conditions correctly capture subtle fairness violations. We then use these conditions as the basis for database repair algorithms that provide provable fairness guarantees about classifiers trained on their training labels. We demonstrate the effectiveness of our proposed techniques with experimental results.

1. INTRODUCTION

Fairness is increasingly recognized as a critical component of machine learning (ML) systems. These systems are now routinely used to make decisions that affect people's lives [7], with the aim of reducing costs, reducing errors, and improving objectivity. While this is a positive trend, there is also enormous potential for harm. The functionality of ML systems are defined by their parameters as dictated by the data used for training them. More often than not, the available data reflects societal inequities and historical biases, and, as a consequence, the models trained on such data will therefore reinforce and legitimize discrimination and opacity.

There has been a steady stream of reports of discriminatory ML systems, due to biased data, across many different domains. In 2014, a team of machine learning experts from Amazon Inc. began work on an automated system to review job applicants' resumes. According to a recent Reuters article [8], the experimental system gave job candidates scores ranging from one to five and was trained on 10 years of recruiting data from Amazon. However, by 2015 the team realized that the system showed a significant gender bias towards male candidates over females due to historical discrimination in the training data. Amazon edited the system to make it gender agnostic, but there was no guarantee that discrimination did not occur through other means, and the project was totally abandoned in 2017.

In another example, in 2016, a team of journalists from ProPublica analysed COMPAS, one of the many widely used commercial risk assessment algorithms for predicting recidivism, and revealed that it overpredicts recidivism for African-Americans and underpredicts it for Caucasians [20]. In the context of predicting recidivism (which is itself a questionable application!), fairness issue arise because these systems are trained using data on arrested individuals, as opposed to data on individuals who commit crime. Because of historical racial biases in arrest data, probabilities produced by these systems are racially biased as well.

Mitigating Bias. These examples underpin the importance of understanding and accounting for historical bias in data. A naïve (and ineffective) approach sometimes used in practice is to simply omit the protected attribute (say, race or gender) when training the classifier. However, since the protected attribute is frequently represented implicitly by some combination of proxy variables, the classifier still learns the discrimination present in training data. For example, zip code tends to predict race due to a history of segregation [13, 34]; answers to personality tests identify people with disabilities [37]; and keywords can reveal gender on a resume [8]. As a result, a classifier trained without regard to the protected attribute not only fails to remove discrimination, but it can complicate the detection and mitigation of discrimination downstream via existing techniques [29, 6, 5, 18, 17, 24, 36], such as those we describe next.

The two main approaches to reduce or eliminate sources of discrimination are summarized in Fig. 1. The most popular is in-processing, where the ML algorithm itself is modified to account for fairness during the training time; this approach must be reimplemented for every ML application. The alternative is to process either the training data (pre-processing) or the output of the classifier itself (post-processing). We advocate for the pre-processing strategy, which can be designed to be agnostic to the choice of ML algorithm and instead interprets the problem as a database repair task.

©ACM 2020. This is a minor revision of the paper entitled "Interventional Fairness: Causal Database Repair for Algorithmic Fairness", published in SIGMOD'19, ISBN978-1-4503-3531-7/16/06, June 26-July 01, 2016, Amsterdam, The Netherlands. DOI: https://doi.org/10.1145/3299869.33199015.

Fairness Definitions. One needs a quantitative measure of discrimination in order to remove it. A large number of fairness definitions have been proposed, which we broadly categorize in Fig. 1. The best-known measures are based on statistical (i.e., *associative*) relationships between the protected attribute and the outcome. For example, *demographic parity* requires that, for all groups of the protected attribute, the overall probability of a positive prediction of an outcome should be the same. However, it has been shown that associative definitions of fairness can be mutually exclusive [5] and fail to distinguish between discriminatory, non-discriminatory, and spurious association between a protected attribute and the outcome of an algorithm [17, 24, 9]. The following example highlights the pitfalls of associative fairness:

Example 1.1. *In 1973, UC Berkeley was sued for discrimination against females in graduate school admissions when it was found that 34.6% of females were admitted in 1973 as opposed to 44.3% of males, hence demographic parity was violated. However, analysis revealed that the effect occurred because females tended to apply to departments with lower overall acceptance rates [30]. When broken down by department, a slight bias* toward *female applicants was observed, a result that did not constitute evidence for gender-based discrimination.*

Such situations have recently motivated a search for a more principled measure of fairness and discrimination based on *causality* [17, 24, 18, 29, 31]. These approaches assume access to background knowledge on the underlying causal models that usually visualised as directed graphs, consisting of nodes (representing variables) and directed edges between the nodes (representing potential causal relations). These approaches, then, measure discrimination as the causal influence of the protected attribute on the outcome of an algorithm, through certain causal paths that deemed to be socially unacceptable. For instance, in Example 1.1, the direct causal influence of gender on admission decisions as well as its indirect effect through applicants' hobbies might be considered as discriminatory. In terms of causal models, the former is expressed by prohibiting the directed edge from gender to admission decision, and the latter is expressed by prohibiting any directed path from gender to hiring decision that is intercepted by applicant's hobbies. However, causal approaches to fairness assume access to a complete causal model, and no existing proposals describe comprehensive systems for pre-processing data to mitigate causal discrimination.

Fairness via Database Repair. This paper describes a new approach to removing discrimination by *repairing the training data*. Our proposal is based on the following key observations: 1) In causal models, a missing arrow between two variables X and Y encodes the assumption that there exists a set of variables $\mathbf{Z}$ such that X and Y are statistically independent given $\mathbf{Z}$; denoted as the conditional independence statement $(X \perp\!\!\!\perp Y \mid \mathbf{Z})$. Consequently, causal fairness constraints (expressed as requirements about the absence of certain causal paths from protected attributes to an outcome) can be compiled into conditional independence statements. Therefore, to enforce causal fairness, we can intervene on the data and enforce the corresponding conditional independence statements instead of intervening on the causal models over which we have no control. 2) There is a clear connection between conditional independence statements and well-studied integrity constraints in data management such as Multivalued Dependencies (MVDs) [1]. Our paper leverages these connections to frame algorithmic fairness as a database repair problem for Multivalued Dependencies. The problem of database repair has been studied for various types of constraints, for example the complexity of repairing for functional dependencies (FD) has been completely solved in [21]. However, the problem of database repairs for MVDs has received less attention and is still open. Recently, the problem of mining MVDs from data is studied in [16].

	Statistical	Causal
In-processing (Modify Algorithm)	[15, 41, 3, 24, 17]	[24, 17, 29]
Pre/post-processing (Modify input/output Data)	[10, 4, 12, 39]	Capuchin (this paper)

Figure 1: Fairness metrics and enforcement methods.

Capuchin. Our system, Capuchin, accepts a dataset consisting of a protected attribute (e.g., gender, race, etc.), an outcome attribute (e.g., college admissions, loan application, or hiring decisions), and a set of *admissible variables* through which it is permissible for the protected attribute to influence the outcome. For instance, the applicant's choice of department in Example 1.1 may be considered as admissible despite being correlated with gender. The system repairs the input data by inserting or removing tuples to remove the influence of the protected attribute on the outcome through any directed causal paths that includes inadmissible attributes, by means of enforcing the corresponding MVDs. That is, the repaired training data can be seen as a *sample from a counterfactual fair world*.

Unlike previous measures of fairness based on causality [24, 17, 29], which require the presence of the underlying causal model, our definition is based solely on the notion of *intervention* [25] and can be guaranteed even in the absence of causal models. The user needs only distinguish admissible and inadmissible attributes; we prove that this information is sufficient to mitigate discrimination.

We use this *interventional* approach to derive in Sec. 3.1 a new fairness definition, called *justifiable fairness*. Justifiable fairness subsumes and improves on several previous definitions and can correctly distinguish fairness violations and non-violations that would otherwise be hidden by statistical coincidences, such as Simpson's paradox. We prove next, in Sec. 3.2, that, if the training data satisfies a simple saturated conditional independence, then any reasonable algorithm trained on it will be fair.

Our core technical contribution consists of a new approach to repair training data in order to enforce the saturated conditional independence that guarantees fairness. In Sec. 4 we first define the problem formally and then present a new technique to reduce it to a multivalued functional dependency MVD [1]. Then, we introduce new techniques to repair a dataset for an MVD. In Sec. 5 we evaluate our algorithms on real data and show that they meet our goals.

2. PRELIMINARIES

This section reviewers the basic background on database repair, algorithmic fairness and causal inference, the building blocks of our paper.

We denote variables (i.e., dataset attributes) by upper-case letters, X, Y, Z, V; their values with lowercase letters, x, y, z, v; and denote sets of variables or values using boldface ($\mathbf{X}$ or $\mathbf{x}$). The domain of a variable X is $Dom(X)$, and the domain of a set of variables is $Dom(\mathbf{X}) = \prod_{Y \in \mathbf{X}} Dom(Y)$. In this paper, all domains are discrete and finite; continuous domains are assumed to be binned, as is typical. A *database instance* D is a relation whose attributes we denote as $\mathbf{V}$. We assume set semantics (*i.e.*, no duplicates) unless otherwise stated, and we denote the cardinality of D as $n = |D|$. Given a partition $\mathbf{X} \cup \mathbf{Y} \cup \mathbf{Z} = \mathbf{V}$, we say that D satisfies the *multivalued dependency* (MVD) $\mathbf{Z} \twoheadrightarrow \mathbf{X}$ if $D = \Pi_{\mathbf{XZ}}(D) \bowtie \Pi_{\mathbf{ZY}}(D)$.

Typically, training data for ML is a bag B. We convert it into a set D (by eliminating duplicates) and a probability distribution Pr, which accounts for multiplicities; We call D the support of Pr. We say that Pr is *uniform* if all tuples have the same probability. We say $\mathbf{X}$ and $\mathbf{Y}$ are *conditionally independent (CI)* given $\mathbf{Z}$, written $(\mathbf{X} \perp\!\!\!\perp_{\Pr} \mathbf{Y} | \mathbf{Z})$, or just $(\mathbf{X} \perp\!\!\!\perp \mathbf{Y} | \mathbf{Z})$, if $\Pr(\mathbf{x}|\mathbf{y}, \mathbf{z}) = \Pr(\mathbf{x}|\mathbf{z})$ whenever $\Pr(\mathbf{y}, \mathbf{z}) > 0$. When $\mathbf{V} = \mathbf{XYZ}$, then the CI is said to be *saturated*. A uniform Pr satisfies a saturated CI iff its support D satisfies the MVD $\mathbf{Z} \twoheadrightarrow \mathbf{X}$. Training data usually does not have a uniform Pr, and in such cases the equivalence between the CI and MVD fails [38]. This issue can be addressed by converting a bag to a corresponding set; see [32] for details.

The *database repair problem* is the following: we are given a set of constraints Γ and a database instance D, and we need to perform a minimal set of updates on D such that the new database D' satisfies Γ [2].

2.1 Background on Algorithmic Fairness

Algorithmic fairness considers a *protected attribute* S, a *response variable* Y, and a prediction algorithm $A : Dom(\mathbf{X}) \to Dom(O)$, where $\mathbf{X} \subseteq \mathbf{V}$, and the prediction of A is denoted O (some references denote it $\tilde{Y}$) and called *outcome*. For simplicity, we assume S classifies the population into protected $S = 1$ and privileged $S = 0$, for example, female and male. Fairness definitions can be classified as statistical or causal.

Statistical Fairness. This family of fairness definitions is based on statistical measures on the variables of interest; a summary is shown in Fig. 2. *Demographic Parity* (DP) [3, 14, 42, 35, 9], requires an algorithm to classify both the protected and the privileged group with the same probability. As we saw in Example 1.1, the lack of statistical parity cannot be considered as evidence for gender-based discrimination; this has motivated the introduction of *Conditional Statistical Parity* (CSP) [6], which controls for a set of admissible factors $\mathbf{A}$. Another popular measure used for predictive classification algorithms is *Equalized Odds* (EO), which requires that both protected and privileged groups to have the same false positive (FP) rate, and the same false negative (FN) rate. Finally, *Predictive Parity* (PP) requires that both protected and unprotected groups have the same predicted positive value (PPV) It has been shown that these measures are inconsistent [5].

Causal Fairness. Causal notions of fairness were motivated by the need to address difficulties generated by statistical fairness and assumes an underlying causal model [18, 17, 24, 29, 11]. We first discuss causal DAGs before reviewing causal fairness.

Fairness Metric	**Description**
Demographic Parity (DP) [9, 35]	$S \perp\!\!\!\perp O$
Conditional Statistical parity [6]	$S \perp\!\!\!\perp O \mid \mathbf{A}$
Equalized Odds (EO) [12, 40]	$S \perp\!\!\!\perp O \mid Y$
Predictive Parity (PP)[5, 35, 5, 12]	$S \perp\!\!\!\perp Y \mid O$

Figure 2: Common statistical definitions of fairness.

2.2 Background on Causal DAGs

Causal DAG. A *causal DAG* G over set of variables $\mathbf{V}$ is a directed acyclic graph that models the functional interaction between variables in $\mathbf{V}$. Each node X represents a variable in $\mathbf{V}$ that is functionally determined by: (a) its parents $\mathbf{Pa}(X)$ in the DAG, and (b) some set of *exogenous* factors that need not appear in the DAG, as long as they are mutually independent. This functional interpretation leads to the same decomposition of the joint probability distribution of $\mathbf{V}$ that characterizes Bayesian networks [25]:

$$\Pr(\mathbf{V}) = \prod_{X \in \mathbf{V}} \Pr(X | \mathbf{Pa}(X)) \tag{1}$$

***d*-Separation and Faithfulness.** A common inference question in a causal DAG is how to determine whether a CI $(\mathbf{X} \perp\!\!\!\perp \mathbf{Y} | \mathbf{Z})$ holds. A sufficient criterion is given by the notion of d-separation, a syntactic condition $(\mathbf{X} \perp\!\!\!\perp \mathbf{Y} |_d \mathbf{Z})$ that can be checked directly on the graph. Pr and G are called *Markov compatible* if $(\mathbf{X} \perp\!\!\!\perp \mathbf{Y} |_d \mathbf{Z})$ implies $(\mathbf{X} \perp\!\!\!\perp_{\Pr} \mathbf{Y} | \mathbf{Z})$; if the converse implication holds, then we say that Pr is *faithful* to G. If G is a causal DAG and Pr is given by Eq.(1), then they are Markov compatible [26].

Counterfactuals and do Operator. A *counterfactual* is an intervention where we actively modify the state of a set of variables $\mathbf{X}$ in the real world to some value $\mathbf{X} = \mathbf{x}$ and observe the effect on some output Y. Pearl [25] described the *do* operator that allows this effect to be computed on a causal DAG, denoted $\Pr(Y | do(X = x))$. To compute this value, we assume that X is determined by a constant function $X = x$ instead of a function provided by the causal DAG. This assumption corresponds to a modified graph with all edges into $\mathbf{X}$ removed, and values of $\mathbf{X}$ are set to $\mathbf{x}$. The Bayesian rule Eq.(1) for the modified graph defines $\Pr(Y | do(\mathbf{X} = \mathbf{x}))$; the exact expression is in [25, Theorem 3.2.2]. We proved and illustrated the following in [32]:

THEOREM 2.1. *Given a causal DAG G and a set of variables $\mathbf{X} \subseteq \mathbf{V}$, suppose $\mathbf{X} = \{X_0, X_1 \ldots, X_m\}$ are ordered such that X_i is a non-descendant of X_{i+1} in G. The effect of a set of interventions $do(\mathbf{X} = \mathbf{x})$ is given by the following* extended adjustment formula*:*

$$\Pr(y | do(\mathbf{X} = \mathbf{x})) = \sum_{\mathbf{z} \in Dom(\mathbf{Z})} \Pr(y | \mathbf{x}, \mathbf{z}) \left(\prod_{i=0}^{m} \Pr\left(\mathbf{pa}(X_i) \middle| \bigcup_{j=0}^{i-1} \mathbf{pa}(X_j), \bigcup_{j=0}^{i-1} x_j \right) \right) \tag{2}$$

where $\mathbf{Z} = \bigcup_{X \in \mathbf{X}} \mathbf{Pa}(X)$ and $j \geq 0$.

2.3 Causal Fairness

Counterfactual Fairness. Kusner et al. [18, 19] (see also the discussion in [22]) define a classifier as *counterfactually fair* if the protected attribute of an individual is not a cause of the outcome of the classifier for that individual, i.e., had the protected attributes of the individual been different, and other things being equal, the outcome of the predictor would have remained the same. However, it is known that individual-level counterfactuals can not be estimated from data in general [26].

Proxy Fairness. To avoid individual-level counterfactuals, a common approach is to study population-level counterfactuals or interventional distributions that capture the effect of interventions at the population level rather than an individual level [26, 27, 28]. Kilbertus et. al. [17] defined proxy fairness as follows:

$$P(\tilde{Y}=1|do(\mathbf{P}=\mathbf{p}))=P(\tilde{Y}=1|do(\mathbf{P}=\mathbf{p}')) \qquad (3)$$

for any $\mathbf{p},\mathbf{p}' \in Dom(\mathbf{P})$, where $\mathbf{P}$ consists of proxies to a sensitive variable S (and might include S). Intuitively, a classifier satisfies proxy fairness in Eq 3, if the distribution of $\tilde{Y}$ under two interventional regimes in which $\mathbf{P}$ set to $\mathbf{p}$ and $\mathbf{p}'$ is the same. Thus, proxy fairness is not an individual-level notion. The next example shows that proxy fairness fails to capture group-level discrimination in general.

Example 2.2. *To illustrate the difference between counterfactual and proxy fairness, consider the following college admission example. Both departments make decisions based on students' gender and qualifications, $O = f(G, D, Q)$, where, O stands for admission decision and G, D and Q are binary variables that respectively stands for applicants' gender, their choice of department and qualifications. The causal DAG is $G \rightarrow O, D \rightarrow O, Q \rightarrow O$. Let $D = U_D$ and $Q = U_Q$, where U_D and U_Q are exogenous factors that are independent and that are uniformly distributed, e.g., $P(U_Q = 1) = P(U_Q = 0) = \frac{1}{2}$. Further suppose $f(G, \text{'A'}, Q) = G \wedge Q$ and $f(G, \text{'B'}, Q) = (1-G) \wedge Q$, i.e., dep. A admits only qualified males and dep. B admits only qualified females. This admission process is proxy-fair, because $P(O = 1|do(G = 1)) = P(O = 1|do(G = 0)) = \frac{1}{2}$. On the other hand, it is clearly individually-unfair, in fact it is group-level unfair (for all applicants to the same department).*

Path-Specific Fairness. These definitions are based on graph properties of the causal graph, *e.g.*, prohibiting specific paths from the sensitive attribute to the outcome [24, 22]; however, identifying path-specific causality from data requires very strong assumptions and is often impractical.

3. DEFINING AND ENFORCING FAIRNESS

In this section we introduce a new definition of fairness, which, unlike proxy fairness [17], correctly captures group-level fairness, and unlike counterfactual fairness [18, 19] is based on the standard notion of intervention and hence is testable from the data. In the next section we will describe how to repair an unfair training dataset to enforce fairness.

3.1 Interventional Fairness

In this section we assume that the causal graph is given. The algorithm computes an output variable O from input variables $\mathbf{X}$ (Sec. 2.1). We begin with a definition describing when an outcome O is causally independent of the protected attribute S for any possible configuration of a given set of variables $\mathbf{K}$.

Definition 3.1 ($\mathbf{K}$-fair). *Fix a set of attributes $\mathbf{K} \subseteq \mathbf{V} - \{S, O\}$. We say an algorithm $\mathcal{A} : Dom(\mathbf{X}) \rightarrow Dom(O)$ is $\mathbf{K}$-fair w.r.t. a protected attribute S if, for any context $\mathbf{K} = \mathbf{k}$ and every outcome $O = o$, the following holds:*

$$\Pr(O=o|do(S=0),do(\mathbf{K}=\mathbf{k}))=\Pr(O=o|do(S=1),do(\mathbf{K}=\mathbf{k})) \quad (4)$$

We call an algorithm *interventionally fair* if it is $\mathbf{K}$-fair for every set $\mathbf{K}$. Unlike proxy fairness, this notion correctly captures group-level fairness, because it ensures that S does not affect O in *any configuration* of the system obtained by fixing other variables at some arbitrary values. Unlike counterfactual fairness, it does not attempt to capture fairness at the individual level, and therefore it uses the standard definition of intervention (the `do`-operator). In fact, we argue that interventional fairness is the strongest notion of fairness that is testable from data, yet correctly captures group-level fairness. We illustrate with an example (see also Ex 3.6).

Example 3.2. *In contrast to proxy fairness, interventional fairness correctly identifies the admission process in Ex. 2.2 as unfair at department-level. This is because the admission process fails to satisfy $\{D\}$-fairness since, $P(O = 1|do(G = 0), do(D = \text{'A'})) = 0$ but $P(O = 1|do(G = 1), do(D = \text{'A'})) = \frac{1}{2}$. Therefore, interventional fairness is a more fine-grained notion than proxy fairness. We note however that, interventional fairness does not guarantee individual fairness in general. To see this suppose the admission decisions in both departments are based on student's gender and an unobserved exogenous factor U_O that is uniformly distributed, i.e., $O = f(G, U_O)$, such that $f(G, 0) = G$ and $f(G, 1) = 1 - G$. Hence, the causal DAG is $G \rightarrow O$. Then the admission process is $\emptyset$-fair because, $P(O = 1|do(G = 1)) = P(O = 1|do(G = 0)) = \frac{1}{2}$. Therefore, it is interventionally fair (since $\mathbf{V} - \{O, G\} = \emptyset$). However, it is clearly unfair at individual level. If the variable U_o were endogenous (i.e. known to the algorithm), then the admission process is no longer interventionally fair, because it is not $\{U_o\}$-fair: $P(O = 1|do(G = 1), do(U_o = 1)) = P(O = 1|G = 1, U_o = 1) = 0$, while $P(O = 1|do(G = 1), do(U_o = 1)) = P(O = 1|G = 0, U_o = 1) = 1$.*

In practice, interventional fairness is too restrictive, as we show below. To make it practical, we allow the user to classify variables into *admissible* and *inadmissible*. The former variables through which it is permissible for the protected attribute to influence the outcome. In Example 1.1, the user would label department as admissible since it is considered a fair use in admissions decisions, and would (implicitly) label all other variables such as hobby as inadmissible. Only users can identify this classification, and therefore admissible variables are part of the problem definition:

Definition 3.3 (Fairness application). *A fairness application over a domain $\mathbf{V}$ is a tuple $(\mathcal{A}, S, \mathbf{A}, \mathbf{I})$, where $\mathcal{A} : Dom(\mathbf{X}) \rightarrow Dom(O)$ is an algorithm mapping input variables $\mathbf{X} \subseteq \mathbf{V}$ to an outcome $O \in \mathbf{V}$, $S \in \mathbf{V}$ is the protected attribute, and $\mathbf{A} \cup \mathbf{I} = \mathbf{V} - \{S, O\}$ is a partition of the variables into admissible and inadmissible.*

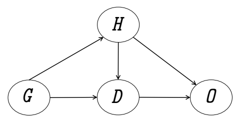
We can now introduce our definition of fairness:

Definition 3.4 (Justifiable fairness). *A fairness application $(\mathcal{A}, S, \mathbf{A}, \mathbf{I})$ is* justifiably fair *if it is $\mathbf{K}$-fair w.r.t. all supersets $\mathbf{K} \supseteq \mathbf{A}$.*

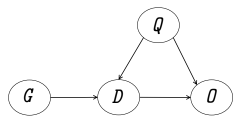
Notice that interventional fairness corresponds to the case where no variable is admissible, i.e., $\mathbf{A} = \emptyset$.

We give next a characterization of justifiable fairness in terms of the structure of the causal DAG:

Here D is not a proxy to G, because $D \perp\!\!\!\perp G$ by assumption.



a) College I



b) College II

College I	Dept. A		Dept. B		Total	
	Admitted	Applied	Admitted	Applied	Admitted	Applied
Male	16	20	16	80	32	100
Female	16	80	16	20	32	100

College II	Dept. A		Dept. B		Total	
	Admitted	Applied	Admitted	Applied	Admitted	Applied
Male	10	10	40	90	50	100
Female	40	50	10	50	50	100

Figure 3: Admission process representation in two colleges where the associational notions of fairness fail (see Ex.3.6).

THEOREM 3.5. *If all directed paths from S to O go through an admissible attribute in $\mathbf{A}$, then the algorithm is justifiably fair. If the probability distribution is faithful to the causal DAG, then the converse also holds.*

To ensure interventional fairness, a sufficient condition is that there exists no path from S to O in the causal graph (because $\mathbf{A} = \emptyset$). Hence, under faithfulness, interventional fairness implies fairness at individual-level, i.e., intervening on the sensitive attribute does not change the counterfactual outcome of individuals. Since this is too strong in most scenarios, we adopt justifiable fairness instead. We illustrate with an example.

EXAMPLE 3.6. *Fig 3 shows how fair or unfair situations may be hidden by coincidences but exposed through causal analysis. In both examples, the protected attribute is gender G, and the admissible attribute is department D. Suppose both departments in College I are admitting only on the basis of their applicants' hobbies. Clearly, the admission process is discriminatory in this college because department A admits 80% of its male applicants and 20% of the female applicants, while department B admits 20% of male and 80% of female applicants. On the other hand, the admission rate for the entire college is the same 32% for both male and female applicants, falsely suggesting that the college is fair. Suppose H is a proxy to G such that $H = G$ (G and H are the same), then proxy fairness classifies this example as fair: indeed, since Gender has no parents in the causal graph, intervention is the same as conditioning, hence* $\Pr(O = 1|do(G = i)) = \Pr(O = 1|G = i)$ *for $i = 0, 1$. Of the previous methods, only conditional statistical parity correctly indicates discrimination. We illustrate how our definition correctly classifies this examples as unfair. Assuming the user labels the department D as admissible, $\{D\}$-fairness fails because, by Eq.(2),* $\Pr(O = 1|do(G = 1), do(D = \text{'A'})) = \sum_h \Pr(O = 1|G = 1, D = \text{'A'}, h)\Pr(h|G = 1) = \Pr(O = 1|G = 1, D = \text{'A'}) = 0.8$, *and, similarly* $\Pr(O = 1|do(G = 0), do(D = \text{'A'})) = 0.2$. *Therefore, the admission process is not justifiably fair.*

Now, consider the second table for College II, where both departments A and B admit only on the basis of student qualifications Q. A superficial examination of the data suggests that the admission is unfair: department A admits 80% of all females, and 100% of all male applicants; department B admits 20% and 44.4% respectively. Upon deeper examination of the causal DAG, we can see that the admission process is justifiably fair because the only path from Gender to the Outcome goes through department, which is an admissible attribute. To understand how the data could have resulted from this causal graph, suppose 50% of each gender have high qualifications and are admitted, while others are rejected, and that 50% of females apply to each department but more qualified females apply to department A than to B (80% v.s. 20%). Further, suppose fewer males apply to department A, but all of them are qualified. The algorithm satisfies demographic parity and proxy fairness but fails to satisfy conditional statistical parity since $\Pr(A = 1|G = 1, D = \text{'A'}) = 0.8$ *but* $\Pr(A = 1|G = 0, D = \text{'A'}) = 0.2$). *Thus, conditioning on D falsely indicates discrimination in College II. One can check that the algorithm is justifiably fair, and thus our definition also correctly classifies this example; for example, $\{D\}$-fairness follows from Eq.(2):* $\Pr(O = 1|do(G = i), do(D = d)) = \sum_q \Pr(O = 1|G = i, d, q))\Pr(q|G = i) = \frac{1}{2}$. *To summarize, unlike previous definitions of fairness, justifiable fairness correctly identifies College I as discriminatory and College II as fair.*

3.2 Testing Fairness on the Training Data

In this section we introduce a sufficient condition for testing justifiable fairness, which uses only the training data $D, \Pr$ (Sec. 2) and does not require access to the causal graph G. We assume only that G and $\Pr$ are Markov compatible (Sec. 2.2). The training data has an additional response variable Y. As before, we assume a fairness application ($\mathcal{A}$, $S, \mathbf{A}, \mathbf{I}$) is given and that the algorithm is a good prediction of the response variable, *i.e.* $\Pr(Y = 1|\mathbf{X} = \mathbf{x}) \approx \Pr(O = 1|\mathbf{X} = \mathbf{x})$; we call the algorithm a *reasonable* classifier to indicate that it satisfies this condition. Note that this is a typical assumption in pre-processing approaches such as [4] and is needed to decouple the the issues of model accuracy and fairness. If the distributions of $\Pr(Y = 1|\mathbf{X} = \mathbf{x})$ and $\Pr(O = 1|\mathbf{X} = \mathbf{x})$ could be arbitrarily far apart, no fairness claims can be made about a classifier that, for example, imposes a pre-determined distribution on the outcome predictions rather than learning an approximation of $\Pr(Y = 1|\mathbf{X} = \mathbf{x})$ from the training data.

We first establish a technical condition for fairness based on the Markov boundary, and then simplify it. Recall that given a probability distribution $\Pr$, the *Markov boundary* of a variable $Y \in \mathbf{V}$, denoted $\mathbf{MB}(Y)$, is a minimal subset of $\mathbf{V} - \{Y\}$ that satisfies the saturated conditional independence $(Y \perp\!\!\!\perp_{\Pr} \mathbf{V} - (\mathbf{MB}(Y) \cup \{Y\})|\mathbf{MB}(Y))$. Intuitively, $\mathbf{MB}(Y)$ shields Y from the influence of other variables. We prove:

THEOREM 3.7. *A sufficient condition for a fairness application $(\mathcal{A}, S, \mathbf{A}, \mathbf{I})$ to be justifiably fair is $\mathbf{MB}(O) \subseteq \mathbf{A}$.*

The condition in Theorem 3.7 can be checked without knowing the causal DAG, but requires the computation of the Markov boundary; moreover, it is expressed in terms of the outcome O of the algorithm. We derive from here a sufficient condition that refers only to the response variable Y present in the training data.

COLOLLARY 3.8. *Fix a training data $D, \Pr$, where $Y \in \mathbf{V}$ is the training label, and $\mathbf{A}, \mathbf{I}$ are admissible and inadmissible attributes. Then any reasonable classifier trained on a set of variables $\mathbf{X} \subseteq \mathbf{V}$ is justifiably fair w.r.t. a protected attribute S, if either: (a)* $\Pr$ *satisfies the CI* $(Y \perp\!\!\!\perp \mathbf{X} \cap \mathbf{I}|\mathbf{X} \cap \mathbf{A})$, *or (b)* $\mathbf{X} \supseteq \mathbf{A}$ *and* $\Pr$ *satisfies the saturated CI* $(Y \perp\!\!\!\perp I|\mathbf{A})$.

While condition (a) is the weaker assumption, condition (b) has the advantage that the CI is saturated. Our method

D:

	X	Y	Z	Pr
t_1	a	a	c	3/8
t_2	a	b	c	2/8
t_3	b	a	c	2/8
t_4	b	b	d	1/8

D_1 :

	X	Y	Z
t_1	a	a	c
t_2	a	b	c
t_3	b	a	c
t_4	b	b	c
t_5	b	b	d

D_2 :

	X	Y	Z
t_1	a	a	c
t_2	a	b	c
t_4	b	b	d

Figure 4: A simple database repair: D does not satisfy the MVD $Z \twoheadrightarrow X$. In D_1, we inserted the tuple (b, b, c) to satisfy the MVD, and in D_2 we deleted the tuple (b, a, c) to satisfy the MVD.

for building a fair classifier is to repair the training data in order to enforce (b).

3.3 Building Fair Classifiers

A naive way to satisfy Corollary 3.8(a) is to set $\mathbf{X} = \mathbf{A}$, in other words to train the classifier only on admissible attributes This method guarantees fairness, but it is impractical and can negatively affect the accuracy of the classifier [32]. Instead, our approach is to repair the training data to enforce the condition in Corollary 3.8(b). We consider the saturated CI $(Y \perp\!\!\!\perp \mathbf{I}|\mathbf{A})$ as an *integrity constraint* that should always hold in training data $D, \Pr$. Capuchin performs a sequence of database updates (insertions and deletions of tuples) to obtain another training database D' to satisfy $(Y \perp\!\!\!\perp \mathbf{I}|\mathbf{A})$. We describe this repair problem in Sec. 4. In terms of the causal DAG, this approach can be seen as modifying the underlying causal model to enforce the fairness constraint. However, instead of intervening on the causal DAG, over which we have no control, we intervene on the data to ensure fairness. *Note that minimal repairs are crucial for preserving the utility of data.*

4. DATA REPAIR TO ENSURE FAIRNESS

We have shown in Corollary 3.8 that, if the training data D satisfies a certain saturated conditional independence (CI), then a classification algorithm trained on $D, \Pr$ is justifiably fair. We show here how to modify (repair) the training data to enforce the CI and thus ensure that any reasonable classifier trained on it will be justifiably fair.

4.1 Minimal Repair for MVD and CI

We first consider repairing an MVD. Fix an MVD $\mathbf{Z} \twoheadrightarrow \mathbf{X}$ and a database D that does not satisfy it. The minimal database repair problem is this: find another database D' that satisfies the MVD such that the distance between D and D' is minimized. In this section, we restrict the distance function to the symmetric difference, i.e, $|\Delta(D, D')|$.

Example 4.1. *Consider the database D in Fig. 4 (ignoring the probabilities for the moment), and the MVD $Z \twoheadrightarrow X$. D does not satisfy the MVD. The figure shows two minimal repairs, D_1, D_2, one obtained by inserting a tuple, and the other by deleting a tuple.*

However, our problem is to repair for a saturated CI, not an MVD, since that is what is required in Corollary 3.8. The repair problem for a database constraint is well-studied in the literature, but here we need to repair to satisfy a CI, which is not a database constraint. We first formally define the repair problem for a CI and then show how to reduce it to the repair for an MVD. More precisely, our input is a database D and a probability distribution Pr, and the goal is to define a "repair" $D', \Pr'$ that satisfies the given CI.

We assume that all probabilities are rational numbers. Let the *bag associated* to $D, \Pr$ be the smallest bag B such that Pr is the empirical distribution on B. In other words, B is obtained by replicating each tuple $t \in D$ a number of times proportional to $\Pr(t)$. If Pr is uniform, then $B = D$.

B:

X	Y	Z
a	a	c
a	a	c
a	a	c
a	b	c
a	b	c
b	a	c
b	a	c
b	b	d

D_B :

K	X	Y	Z
1	a	a	c
2	a	a	c
3	a	a	c
1	a	b	c
2	a	b	c
1	b	a	c
2	b	a	c
1	b	b	d

D'_B :

K	X	Y	Z
1	a	a	c
2	a	a	c
1	a	b	c
2	a	b	c
1	b	a	c
1	b	b	c
1	b	b	d

D' :

X	Y	Z	Pr′
a	a	c	2/7
a	b	c	2/7
b	a	c	1/7
b	b	c	1/7
b	b	d	1/7

Figure 5: Repairing a conditional independence (CI).

Definition 4.2. *The minimal repair of $D, \Pr$ for a saturated CI $(\mathbf{X}; \mathbf{Y}|\mathbf{Z})$ is a pair $D', \Pr'$ such that $\Pr'$ satisfies the CI and $|\Delta(B, B')|$ is minimized, where B and B' are the bags associated with $D, \Pr$ and $D', \Pr'$, respectively.*

Recall that $\mathbf{V}$ denotes the set of attributes of D. Let Pr be any probability distribution on the variables $\{K\} \cup \mathbf{V}$, where K is a fresh variable not in $\mathbf{V}$.

Lemma 4.3. *If* Pr *satisfies $(K\mathbf{X}; \mathbf{Y}|\mathbf{Z})$, then it also satisfies $(\mathbf{X}; \mathbf{Y}|\mathbf{Z})$.*

We now describe our method for computing a minimal repair of $D, \Pr$ for some saturated CI. First, we compute the bag B associated to $D, \Pr$. Next, we add the new attribute K to the tuples in B and assign distinct values to $t.K$ to all duplicate tuples t, thus converting B into a set D_B with attributes $K \cup \mathbf{V}$. Importantly, we use as few distinct values for K as possible, i.e., we enumerate the instances of each unique tuple. More precisely, we define:

$$D_B = \{(i, t) \mid t \in B, i = 1, \ldots, |t_B|\} \tag{5}$$

were $|t_B|$ denotes the number of occurrences (or multiplicity) of a tuple t in the bag B. Then, we repair D_B w.r.t. to the MVD $\mathbf{Z} \twoheadrightarrow K\mathbf{X}$, obtaining a repaired database D'_B. Finally, we construct a new training set $D' = \Pi_{\mathbf{V}}(D'_B)$, with the probability distribution obtained by marginalizing the empirical distribution on D'_B to the variables $\mathbf{V}$.

Example 4.4. *Fig 4 shows two repairs D_1 and D_2 of the database D, in Example 4.1, w.r.t the MVD $Z \twoheadrightarrow X$. Consider now the probability distribution,* Pr *shown in the figure. Suppose we want to repair it for the CI $(X; Y|Z)$. Clearly, both D_1 and D_2, when endowed with the empirical distribution* do *satisfy this CI, but they are very poor repairs because they completely ignore the probabilities in the original training data, which are important signals for learning. Our definition captures this by insisting that the repaired bag B' be close to the bag B associated to D,* Pr *(see B in Fig. 5), but the sets D_1 and D_2 are rather far from B. Instead, our method first converts B into a set D_B by adding a new attribute K (see Fig. 5) then, it repairs D_B for the MVD $Z \twoheadrightarrow KX$, obtaining D'_B. The final repair $D', \Pr'$ consists of the empirical distribution on D'_B, but with the attribute K and duplicates removed.*

The problem of computing minimal repairs for MVDs and CIs, as introduced in this section, is essentially an optimization problem. A suit of techniques for addressing these problems has been introduced in [33, 32] that exploit reduction to the MaxSAT and Matrix Factorization.

5. EXPERIMENTAL RESULTS

This section presents experiments that evaluate the feasibility and efficacy of Capuchin. We aim to evaluate the end-to-end performance of Capuchin in terms of utility and fairness, with respect to our repair method. We refer the reader to [32] for more experiments.

5.1 Setup

We report the empirical utility of each classifier using Accuracy (ACC) via 5-fold cross-validation. We evaluate using three classifiers: Linear Regression (LR), Multi-layer Perceptron (MLP), and Random Forest (RF).

To assess the effectiveness of the proposed approaches, we used the ratio of observational discrimination (ROD) defined in [32] as follows: Given a fairness application ($\mathcal{A}$, $S, \mathbf{A}, \mathbf{I}$), let $\mathbf{A}_b = \mathbf{MB}(O) - \mathbf{I}$. We quantify the *ratio of observational discrimination (ROD)* of $\mathcal{A}$ against S in a context $\mathbf{A}_b = \mathbf{a}_b$ as $\delta(S;O|\mathbf{a}_b) \overset{\text{def}}{=} \frac{\Pr(O=1|S=0,\mathbf{a}_b)\Pr(O=0|S=1,\mathbf{a}_b)}{\Pr(O=0|S=0,\mathbf{a}_b)\Pr(O=1|S=1,\mathbf{a}_b)}$. Intuitively, ROD calculates the effect of membership in a protected group on the odds of the positive outcome of $\mathcal{A}$ for subjects that are similar on $\mathbf{A}_b = \mathbf{a}_b$ ($\mathbf{A}_b$ consists of admissible attributes in the Markov boundary of the outcome). ROD is sensitive to the choice of a context $\mathbf{A}_b = \mathbf{a}_b$ by design. The overall ROD denoted by $\delta(S, O|\mathbf{A}_b)$ can be computed by averaging $\delta(S, O|\mathbf{a}_b)$ for all $\mathbf{a}_b \in \mathbf{A}_b$.

5.2 End-To-End Results

In the following experiments, a fairness constraint was enforced on training data using Capuchin repair algorithms (cf. Sec 4). Specifically, each dataset was split into five training and test datasets. All training data were repaired separately using Matrix Factorization (MF), Independent Coupling (IC) and two versions of the MaxSAT approach (see [32] for details of MF and IC methods): MS(Hard), which feeds all clauses of the lineage of a CI into MaxSAT, and MS(Soft), which only feeds small fraction of the clauses. We tuned MaxSAT to enforce CIs approximately. We then measured the utility and discrimination metrics for each repair method as explained in Sec 5.1. For all datasets, the chosen training variables included the Markov boundary of the outcome variables, which were learned from data using the Grow-Shrink algorithm [23] and permutation test [30].

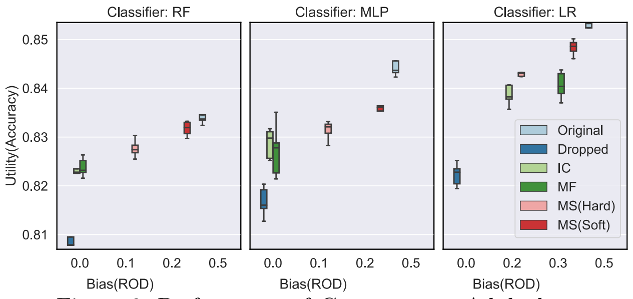


Figure 6: Performance of Capuchin on Adult data.

Adult data. This data reflects historical income inequality that can be reinforced by ML algorithms. We used Capuchin to remove the mentioned sources of discrimination from Adult data. Specifically, we categorized the attributes in the Adult dataset as follows: ($\mathbf{S}$) sensitive attributes: gender (male, female); ($\mathbf{A}$) admissible attributes: hours per week, occupation, age, education, etc.; ($\mathbf{N}$) inadmissible attributes: marital status; (Y) binary outcome: high income. As is common in the literature, we assumed that the potential influence of gender on income through some or all of the admissible variables was fair; However, the direct influence of gender on income, as well as its indirect influence on income through marital status, were assumed to be discriminatory. To remove the bias, we enforced the CI ($Y \perp\!\!\!\perp \mathbf{S}, \mathbf{N}|\mathbf{D}$) on training datasets using the Capuchin repair algorithms. Then, we trained the classifiers on both original and repaired training datasets using the set of variables $\mathbf{A} \cup \mathbf{N} \cup \mathbf{S}$. We also trained the classifiers on original data using only $\mathbf{A}$, i.e., we dropped the sensitive and inadmissible variables.

Fig. 6 compares the utility and bias of Capuchin repair methods on Adult data. As shown, our repair methods delivered surprisingly good results: when partially repairing data using the MaxSAT approach, i.e, using MS(Soft), almost 50% of the bias was removed while accuracy decreased by only 1%.

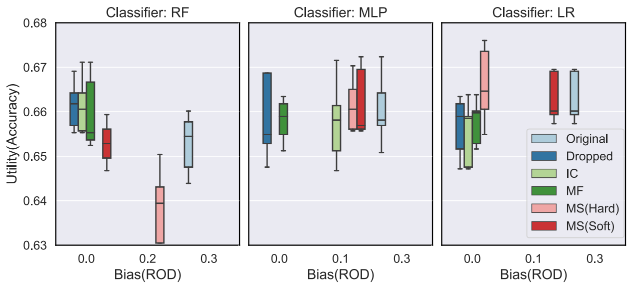


Figure 7: Performance of Capuchin on COMPAS data.

COMPAS. For the second experiment, we used the ProPublica COMPAS dataset [20]. This dataset contains records for all offenders in Broward County, Florida in 2013 and 2014. We categorized the attributes in COMPAS data as follows: ($\mathbf{S}$) protected attributes: race (African American, Caucasian); ($\mathbf{A}$) admissible attributes: number of prior convictions, severity of charge degree, age; **(Y)** binary outcome: a binary indicator of whether the individual is a recidivist. As is common in the literature, we assumed that it was fair to use the admissible attributes to predict recidivism even though they can potentially be influenced by race, and our only goal in this experiment was to address the direct influence of race. We pursued the same steps as explained in the first experiment. Fig. 7 compares the bias and utility of Capuchin repair methods to original data. As shown, all repair methods successfully reduced the ROD. However, we observed that MF and IC performed better than MS on COMPAS data (as opposed to Adult data).

6. CONCLUSIONS

We considered a causal approach for fair ML, reducing it to a database repair problem. We showed that conventional associational and causal fairness metrics can over- and under-report discrimination. We defined a new notion of fairness, called *justifiable fairness*, that addresses shortcomings of the previous definitions and argued that it is the strongest notion of fairness that is testable from data. We then proved sufficient properties for justifiable fairness and use these results to translate the properties into saturated conditional independence that we can be seen as multivalued dependencies with which to repair the data. We then proposed multiple algorithms for implementing these repairs. Our experimental results show that our algorithms successfully mitigate discrimination due to biased training data, are robust to unseen test data.

7. REFERENCES

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] Leopoldo E. Bertossi. *Database Repairing and Consistent Query Answering*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2011.

[3] Toon Calders and Sicco Verwer. Three naive bayes approaches for discrimination-free classification. *Data Mining and Knowledge Discovery*, 21(2):277–292, 2010.

[4] Flavio Calmon, Dennis Wei, Bhanukiran Vinzamuri, Karthikeyan Natesan Ramamurthy, and Kush R Varshney. Optimized pre-processing for discrimination prevention. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 3992–4001. Curran Associates, Inc., 2017.

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

[6] Sam Corbett-Davies, Emma Pierson, Avi Feller, Sharad Goel, and Aziz Huq. Algorithmic decision making and the cost of fairness. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 797–806. ACM, 2017.

[7] Rachel Courtland. Bias detectives: the researchers striving to make algorithms fair. *Nature*, 558, 2018.

[8] Jeffrey Dastin. Rpt-insight-amazon scraps secret ai recruiting tool that showed bias against women. *Reuters*, 2018.

[9] Cynthia Dwork, Moritz Hardt, Toniann Pitassi, Omer Reingold, and Richard Zemel. Fairness through awareness. In *Proceedings of the 3rd innovations in theoretical computer science conference*, pages 214–226. ACM, 2012.

[10] Michael Feldman, Sorelle A Friedler, John Moeller, Carlos Scheidegger, and Suresh Venkatasubramanian. Certifying and removing disparate impact. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 259–268. ACM, 2015.

[11] Sainyam Galhotra, Yuriy Brun, and Alexandra Meliou. Fairness testing: testing software for discrimination. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 498–510. ACM, 2017.

[12] Moritz Hardt, Eric Price, Nati Srebro, et al. Equality of opportunity in supervised learning. In *Advances in neural information processing systems*, pages 3315–3323, 2016.

[13] David Ingold and Spencer Soper. Amazon doesn't consider the race of its customers. should it? *Bloomberg*, 2016. `www.bloomberg.com/graphics/2016-amazon-same-day/`.

[14] Faisal Kamiran and Toon Calders. Classifying without discriminating. In *Computer, Control and Communication, 2009. IC4 2009. 2nd International Conference on*, pages 1–6. IEEE, 2009.

[15] Toshihiro Kamishima, Shotaro Akaho, Hideki Asoh, and Jun Sakuma. Fairness-aware classifier with prejudice remover regularizer. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 35–50. Springer, 2012.

[16] Batya Kenig, Pranay Mundra, Guna Prasaad, Babak Salimi, and Dan Suciu. Mining approximate acyclic schemes from relations. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, June 14-19, 2020*, pages 297–312. ACM, 2020.

[17] Niki Kilbertus, Mateo Rojas Carulla, Giambattista Parascandolo, Moritz Hardt, Dominik Janzing, and Bernhard Schölkopf. Avoiding discrimination through causal reasoning. In *Advances in Neural Information Processing Systems*, pages 656–666, 2017.

[18] Matt J Kusner, Joshua Loftus, Chris Russell, and Ricardo Silva. Counterfactual fairness. In *Advances in Neural Information Processing Systems*, pages 4069–4079, 2017.

[19] Matt J. Kusner, Joshua R. Loftus, Chris Russell, and Ricardo Silva. Counterfactual fairness. *CoRR*, abs/1703.06856, 2017.

[20] Jeff Larson, Surya Mattu, Lauren Kirchner, and Julia Angwin. How we analyzed the compas recidivism algorithm. *ProPublica (5 2016)*, 9, 2016.

[21] Ester Livshits, Benny Kimelfeld, and Sudeepa Roy. Computing optimal repairs for functional dependencies. *ACM Transactions on Database Systems (TODS)*, 45(1):1–46, 2020.

[22] Joshua R Loftus, Chris Russell, Matt J Kusner, and Ricardo Silva. Causal reasoning for algorithmic fairness. *arXiv preprint arXiv:1805.05859*, 2018.

[23] Dimitris Margaritis. Learning bayesian network model structure from data. Technical report, Carnegie-Mellon Univ Pittsburgh Pa School of Computer Science, 2003.

[24] Razieh Nabi and Ilya Shpitser. Fair inference on outcomes. In *Proceedings of the... AAAI Conference on Artificial Intelligence. AAAI Conference on Artificial Intelligence*, volume 2018, page 1931. NIH Public Access, 2018.

[25] Judea Pearl. *Causality*. Cambridge university press, 2009.

[26] Judea Pearl et al. Causal inference in statistics: An overview. *Statistics Surveys*, 3:96–146, 2009.

[27] Donald B Rubin. *The Use of Matched Sampling and Regression Adjustment in Observational Studies*. Ph.D. Thesis, Department of Statistics, Harvard University, Cambridge, MA, 1970.

[28] Donald B Rubin. Statistics and causal inference: Comment: Which ifs have causal answers. *Journal of the American Statistical Association*, 81(396):961–962, 1986.

[29] Chris Russell, Matt J Kusner, Joshua Loftus, and Ricardo Silva. When worlds collide: integrating different counterfactual assumptions in fairness. In *Advances in Neural Information Processing Systems*, pages 6414–6423, 2017.

[30] Babak Salimi, Johannes Gehrke, and Dan Suciu. Bias in olap queries: Detection, explanation, and removal. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1021–1035. ACM, 2018.

[31] Babak Salimi, Harsh Parikh, Moe Kayali, Lise Getoor, Sudeepa Roy, and Dan Suciu. Causal relational learning. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, June 14-19, 2020*, pages 241–256. ACM, 2020.

[32] Babak Salimi, Luke Rodriguez, Bill Howe, and Dan Suciu. Capuchin: Causal database repair for algorithmic fairness. *arXiv preprint arXiv:1902.08283*, 2019.

[33] Babak Salimi, Luke Rodriguez, Bill Howe, and Dan Suciu. Interventional fairness: Causal database repair for algorithmic fairness. In *Proceedings of the 2019 International Conference on Management of Data*, pages 793–810. ACM, 2019.

[34] Andrew D Selbst. Disparate impact in big data policing. *Ga. L. Rev.*, 52:109, 2017.

[35] Camelia Simoiu, Sam Corbett-Davies, Sharad Goel, et al. The problem of infra-marginality in outcome tests for discrimination. *The Annals of Applied Statistics*, 11(3):1193–1216, 2017.

[36] Michael Veale, Max Van Kleek, and Reuben Binns. Fairness and accountability design needs for algorithmic support in high-stakes public sector decision-making. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, pages 440:1–440:14, 2018.

[37] Lauren Weber and Elizabeth Dwoskin. Are workplace personality tests fair? *Wall Strreet Journal*, 2014.

[38] SK Michael Wong, Cory J. Butz, and Dan Wu. On the implication problem for probabilistic conditional independency. *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans*, 30(6):785–805, 2000.

[39] Blake Woodworth, Suriya Gunasekar, Mesrob I. Ohannessian, and Nathan Srebro. Learning non-discriminatory predictors. In *Proceedings of the 2017 Conference on Learning Theory*, pages 1920–1953, 2017.

[40] Muhammad Bilal Zafar, Isabel Valera, Manuel Gomez Rodriguez, and Krishna P Gummadi. Fairness beyond disparate treatment & disparate impact: Learning classification without disparate mistreatment. In *Proceedings of the 26th International Conference on World Wide Web*, pages 1171–1180, 2017.

[41] Muhammad Bilal Zafar, Isabel Valera, Manuel Gomez Rogriguez, and Krishna P. Gummadi. Fairness Constraints: Mechanisms for Fair Classification. In *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics*, pages 962–970, 2017.

[42] Rich Zemel, Yu Wu, Kevin Swersky, Toni Pitassi, and Cynthia Dwork. Learning fair representations. In *International Conference on Machine Learning*, pages 325–333, 2013.

Technical Perspective: Declarative Recursive Computation on an RDBMS

Matthias Boehm
Graz University of Technology
m.boehm@tugraz.at

From a historical perspective, relational database management systems (RDBMSs) have integrated many specialized systems and data models back into the RDBMS over time. New workloads motivated specialized systems for performance, but over time, general-purpose RDBMSs absorbed this functionality to avoid boundary crossing. We already witnessed this process for object-relational functionality, XML and JSON data types, OLAP/HTAP systems, and RDF/graph processing, while for natural language processing (NLP), time series, and machine learning (ML), the outcomes remain unclear. Interestingly, graph processing, NLP, and time series are largely ML workloads too. For this reason, integrating data management and ML is of high practical relevance and has been addressed by (1) integrating ML into RDBMSs, and (2) specialized ML systems. The paper "Declarative Recursive Computation on an RDBMS" [3] by Jankov et al. makes a very valuable contribution by reconciling these two areas and showing the potential of recursive computations on an RDBMS, as the backend—not necessarily frontend—for large-scale machine learning.

ML in RDBMSs: Integrating ML primitives for model training and prediction into RDBMSs has been a major focus area in research and practice for over a decade. Compelling key arguments are to bring analysis close to the data, declarative specification, support for mixed ML and query workloads, scalability, and reuse of existing technology. Existing work includes SQL- and UDF-based systems, factorized learning over joins, deep system integration approaches (e.g., for time series forecasting), tailor-made array databases, efficient and zero-copy data transfers, as well as extended data models and operations. Another long standing question is the effective combination of relational and linear algebra, which recently has been addressed from both systems and theory perspectives as well.

DM in ML Systems: A second major avenue of combining data management (DM) and ML systems is the integration and use of DM techniques—such as rewrites, physical operators, size propagation, query compilation, distributed and federated query processing, caching, incremental maintenance, partitioning, indexing, and compression—into modern ML systems. While such ML systems also leverage data flow graphs at their core, they offer more specialized, stateless domain-specific languages and operators, and directly process files or in-memory matrices or tensors.

Paper Context: The paper by Jankov et al. follows a line of influential work on large-scale statistical computation with SimSQL [1], a distributed RDBMS on Hadoop MapReduce. In this context, the paper reuses the authors' previous work on (1) a chunked matrix representation of blocks with fixed logical size (e.g., $1\text{K} \times 1\text{K}$, whose guaranteed alignment simplifies join processing), integrated as matrix and vector data types [4], as well as (2) ideas on recursive computation of variable dependencies in BUDS [2]. This foundation is interesting because it closely resembles distributed matrix representations of specialized ML systems on distributed computing frameworks like Spark, where lazy evaluation can similarly "unroll" loops into recursive computations.

Paper Contributions: A major problem, however, is the mapping of mini-batch ML training to such recursive computations. The paper by Jankov et al. addresses this problem by two central contributions. First, a succinct SQL extension allows accessing recursive versions of tables via array indexes in a declarative, data–independent manner that facilitates optimization. Such a query is then compiled into a single DAG of relational algebra operations. Second, the operator DAG is partitioned into—independent and thus, manageable—frames by minimizing materialization points, subject to a maximum number of operators per frame. Experiments with large mini-batches and models show very promising scalability results compared to TensorFlow. Overall, this paper has potential for high impact in multiple areas: (1) inspiring improved handling of recursion, DAGs, and large plans in RDBMSs, (2) inspiring frame-based execution in systems with lazy evaluation, and (3) reconciling the diverging areas of In-RDBMS ML and specialized ML systems.

1. REFERENCES

[1] Z. Cai, Z. Vagena, L. L. Perez, S. Arumugam, P. J. Haas, and C. M. Jermaine. Simulation of Database-Valued Markov Chains Using SimSQL. In *SIGMOD*, 2013.

[2] Z. J. Gao, S. Luo, L. L. Perez, and C. Jermaine. The BUDS Language for Distributed Bayesian Machine Learning. In *SIGMOD*, 2017.

[3] D. Jankov, S. Luo, B. Yuan, Z. Cai, J. Zou, C. Jermaine, and Z. J. Gao. Declarative Recursive Computation on an RDBMS. *PVLDB*, 12(7), 2019.

[4] S. Luo, Z. J. Gao, M. N. Gubanov, L. L. Perez, and C. M. Jermaine. Scalable Linear Algebra on a Relational Database System. In *ICDE*, 2017.

Declarative Recursive Computation on an RDBMS

or, Why You Should Use a Database For Distributed Machine Learning

Dimitrije Jankov[†], Shangyu Luo[†], Binhang Yuan[†], Zhuhua Cai*, Jia Zou[‡], Chris Jermaine[†], Zekai J. Gao[†]

Rice University [†]* Arizona State University [‡]

{dj16, sl45, by8, cmj4, jacobgao}@rice.edu [†]

jia.zou@asu.edu [‡] caizhua@gmail.com *

ABSTRACT

We explore the close relationship between the tensor-based computations performed during modern machine learning, and relational database computations. We consider how to make a very small set of changes to a modern RDBMS to make it suitable for distributed learning computations. Changes include adding better support for recursion, and optimization and execution of very large compute plans. We also show that there are key advantages to using an RDBMS as a machine learning platform. In particular, DBMS-based learning allows for trivial scaling to large data sets and especially large models, where different computational units operate on different parts of a model that may be too large to fit into RAM.

1. INTRODUCTION

Modern machine learning (ML) platforms such as TensorFlow [6] have primarily been designed to support *data parallelism*, where a set of almost-identical computations (such as the computation of a gradient) are executed in parallel over a set of computational units. The only difference among the computations is that each operates over different training data (known as "batches"). After each computation has finished, the local gradients are either loaded to a parameter server (in the case of asynchronous data parallelism [17]) or are globally aggregated and used to update the model (in the case of synchronous data parallelism [10]).

Unfortunately, data parallelism has its limits. For example, data parallelism implicitly assumes that the model being learned (as well as intermediate data produced when a batch is used to update the model) can fit in the RAM of a computational unit (which may be a server machine or a GPU). This is not always a reasonable assumption, however. For example, a state-of-the-art NVIDIA Tesla V100 Tensor Core GPU (a $10,000 data center GPU) has 32GB of RAM. 32GB of RAM cannot store the matrix required for a fully-connected layer to encode a vector containing entries from 200,000 categories into a vector of 50,000 neurons. Depending upon the application, 50,000 neurons may not be a lot [19].

The original version of this paper is entitled "Declarative recursive computation on an RDBMS: or, why you should use a database for distributed machine learning" and was published in (Proceedings of the VLDB Endowment, 2019, VLDB Endowment.)

Handling such a model requires *model parallelism*—where the statistical model being learned is not simply replicated at different computational units, but is instead partitioned and operated over in parallel, in a series of bulk-synchronous operations. As discussed in the related work section, systems for distributed ML offer limited support for model parallelism.

Re-purposing relational technology for ML. We argue that if relational technology is used, distinctions such as model vs. data parallelism become unimportant. Relational database management systems (RDBMSs) provide a declarative programming interface, which means that the programmer (or automated algorithm generator, if a ML algorithm is automatically generated via automatic differentiation) only needs to specify what he/she/it wants, but does not need to write out how to compute it. The computations will be automatically generated by the system, and then be optimized and executed to match the data size, layout, and the compute hardware. The code is the same whether the computation is run on a local machine or in a distributed environment, over a small or large model. In contrast, systems such as TensorFlow provide relatively weak forms of declarativity, as each logical operation in a compute graph (such as a matrix multiply) must be executed on some physical compute unit, like a GPU.

Our Contributions. We explore the close relationship between the tensor-based computations performed during modern ML, and relational database computations. We argue that it is easy to express such computations relationally, and detail some of the changes that need to be made to relational systems to support such computations. We show that a lightly-modified (and low-performance) research-prototype relational system can support declarative codes that scale to large model sizes, past those that a platform such as TensorFlow can easily support, and sometimes even outperform TensorFlow on those computations. As larger and larger models and data sets become more prevalent in deep learning (consider the current emphasis on learning huge transformer models [21]), this suggests that tomorrow's high-performance deep learning systems might ideally be based upon relational technology.

2. DEEP LEARNING ON AN RDBMS

2.1 Imperative Programming is Problematic

Imperative programming has been the dominant programming paradigm since the 1950's. In imperative programming, a programmer gives a sequence of commands that incrementally update the state of the program's data. In contrast, since the 1980's relational database codes are almost always written *declaratively*, in SQL.

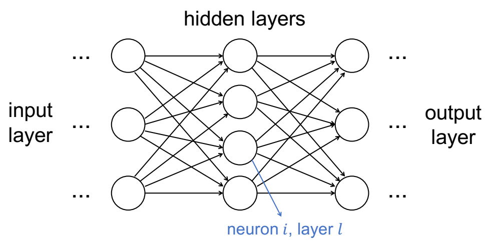


Figure 1: Structure of a feed-forward neural network.

That is, the programmer describes the desired result, ignoring program state, data movement and access, and control flow. Declarative programming is particularly important in databases because if computations are executed suboptimally—if the wrong join order is chosen, for example—they have the potential to produce huge intermediate results that can result in long runtimes or system failure. Before declarative database programming became commonplace, programmers writing imperative database codes proved unable to consistently write programs that would choose the correct data access path. And, even if they wrote the perfect code, the data, storage, or hardware would change, and the code would quickly become obsolete.

Crucially, the tensor-based ML computations performed in deep learning are similar to database computations, in that the same computation can be executed in different ways, and those different execution choices can result in radically different costs. In fact, relations are closely related to tensors—a relation can be viewed as a possible implementation of a tensor—and it is easy to translate computations expressed in standard tensor calculus, such as the Einstein notation [22], into relational joins and aggregations. The key benefit of expressing such computations relationally is that then, in theory, the same code can be executed using a relational optimization and execution engine, regardless of model and data sizes or hardware.

In the remainder of this section, we describe via an example how a simple deep learner can be expressed relationally, and use this example to illustrate the danger of asking a programmer to provide control flow.

2.2 A Simple Deep Learner

A deep neural network is a differentiable, non-linear function, typically conceptualized as a directed graph. Each node in the graph (often called a "neuron") computes a continuous activation function over its inputs (sigmoid, ReLU, etc.).

One of the simplest and most commonly used artificial neural networks is a so-called *feed-forward neural network* [11]. Neurons are organized into layers. Neurons in one layer are connected only to neurons in the next layer, hence the name "feed-forward". Consider the feed-forward network in Figure 1. To compute a function over an input (such as a text document or an image), the input vector is fed into the first layer, and the output from that layer is fed through one or more hidden layers, until the output layer is reached. If the output of layer $l-1$ (or "activation") is represented as a vector $\mathbf{a}_{l-1}$, then the output of layer l is computed as $\mathbf{a}_l = \sigma\left(\mathbf{a}_{l-1}\mathbf{W}_l + \mathbf{b}_l\right)$ Here, $\mathbf{b}_l$ and $\mathbf{W}_l$ are the the bias vector and the weight matrix associated with the layer l, respectively, and $\sigma(\cdot)$ is the activation function.

Learning. *Learning* is the process of customizing the weights for a particular data set and task. Since learning is by far the most computationally intensive part of using a deep network, and because the various data structures (such as the $\mathbf{W}_l$ matrix) can be huge, this is the part we would typically like to distribute across machines.

Two-pass mini-batch gradient descent is the most common learning method used with such networks. Each iteration takes as input the current set of weight matrices $\{\mathbf{W}_1^{(i)}, \mathbf{W}_2^{(i)}, ...\}$ and bias vectors $\{\mathbf{b}_1^{(i)}, \mathbf{b}_2^{(i)}, ...\}$ and then outputs the next set of weight matrices $\{\mathbf{W}_1^{(i+1)}, \mathbf{W}_2^{(i+1)}, ...\}$ and bias vectors $\{\mathbf{b}_1^{(i+1)}, \mathbf{b}_2^{(i+1)}, ...\}$. This process is repeated until convergence.

In one iteration of the gradient descent, each batch of inputs is used to power two passes: the forward pass and the backward pass.

The forward pass. In the forward pass, at iteration i, a small subset of the training data are randomly selected and stored in the matrix $\mathbf{X}^{(i)}$. The activation matrix for each of these data points, $\mathbf{A}_1$, is computed as $\mathbf{A}_1^{(i)} = \sigma\left(\mathbf{X}^{(i)}\mathbf{W}_1^{(i)} + \mathbf{B}_1^{(i)}\right)$ (here, let the bias matrix $\mathbf{B}_1^{(i)}$ be the matrix formed by replicating the bias vector $\mathbf{b}_1^{(i)}$ n times, where n is the size of the mini-batch). Then, this activation is pushed through the network by repeatedly performing the computation $\mathbf{A}_l^{(i)} = \sigma\left(\mathbf{A}_{l-1}^{(i)}\mathbf{W}_l^{(i)} + \mathbf{B}_l^{(i)}\right)$.

The backward pass. At the end of the forward pass, a loss (or error function) comparing the predicted set of values to the actual labels from the training data are computed. To update the weights and biases using gradient descent, the errors are fed back through the network, using the chain rule. Specifically, the errors back-propagated from hidden layer $l+1$ to layer l in the i-th backward pass is computed as

$$\mathbf{E}_l^{(i)} = \left(\mathbf{E}_{l+1}^{(i)}\left(\mathbf{W}_{l+1}^{(i)}\right)^T\right) \odot \sigma'\left(\mathbf{A}_l^{(i)}\right),$$

where $\sigma'(\cdot)$ is the derivative of the activation function. After we have obtained the errors (that serve as the gradients) for each layer, we update the weights and biases:

$$\mathbf{W}_l^{(i)} = \mathbf{W}_l^{(i-1)} - \alpha \cdot \mathbf{A}_{l-1}^{(i-1)}\mathbf{E}_l^{(i-1)},$$

$$\mathbf{b}_l^{(i)} = \mathbf{b}_l^{(i-1)} - \alpha \cdot \sum_n \mathbf{e}_l^{(i-1)},$$

where α is the learning rate, and $\mathbf{e}_l$ is the row vector of $\mathbf{E}_l$.

2.3 A Mixed Imperative/Declarative Approach

Perhaps surprisingly, a model parallel computation of this algorithm is possible on top of an RDBMS. We begin by assuming an RDBMS that has been lightly augmented to handle `matrix` and `vector` data types as described in [16], and assume that the various matrices and vectors have been "chunked". The following database table that stores the chunk of $\mathbf{W}_{\texttt{LAYER}}^{(\texttt{ITER})}$ at the given row and column:

```
W (ITER, LAYER, ROW, COL, MAT)
```

`MAT` is of type `matrix (1000, 1000)` and stores one "chunk" of $\mathbf{W}_{\texttt{LAYER}}^{(\texttt{ITER})}$. A $10^5 \times 10^5$ matrix chunked in this way would have 10^4 entries in the `W` table, with one sub-matrix for each of the $100 = 10^5/10^3$ possible `ROW` values combined with each of the $100 = 10^5/10^3$ possible `COL` values.

Also, the activations $\mathbf{A}_{\texttt{LAYER}}^{(\texttt{ITER})}$ are stored chunked as matrices having 1000 columns in the following table:

```
A (ITER, LAYER, COL, ACT)
```

A final table `AEW` stores the values needed to compete $\mathbf{W}_{\texttt{LAYER}}^{(\texttt{ITER+1})}$: $\mathbf{A}_{\texttt{LAYER-1}}^{(\texttt{ITER-1})}$ (as `ACT`), $\mathbf{E}_{\texttt{LAYER}}^{(\texttt{ITER-1})}$ (as `ERR`), and $\mathbf{W}_{\texttt{LAYER}}^{(\texttt{ITER-1})}$ (as `MAT`):

```
AEW (LAYER, ROW, COL, ACT, ERR, MAT)
```

```
--First, issue a query that computes the errors
--being backpropagated from the top layer in
--the network.
SELECT 9, W.ROW, W.COL, A.ACT, E.ERR, W.MAT
BULK COLLECT INTO AEW
FROM A, W,
  --Note: we are using cross-entropy loss
 (SELECT A.COL,
         crossentropyderiv(A.ACT, DO.VAL) AS ERR
  FROM A, DATA_OUTPUT AS DO
  WHERE A.LAYER=9) AS E
WHERE A.COL=W.ROW AND W.COL=E.COL
  AND A.LAYER=8 AND W.LAYER=9
  AND A.ITER=i AND W.ITER=i;

--Now, loop back through the layers in the network
for l = 9, ..., 2:
  --Use the errors to compute the new weights
  --connecting layer l to layer l + 1; add to
  --result for learning iteration i + 1
  SELECT i+1, l, ROW, COL,
         MAT - matmul(t(ACT), ERR) * 0.00000001
  BULK COLLECT INTO W
  FROM AEW WHERE LAYER=l;

  --Issue a new query that uses the errors from the
  --previous layer to compute the errors in this
  --layer. reluderiv takes the derivative of the
  --activation.
  SELECT l-1, W.ROW, W.COL, A.ACT, E.ERR, W.MAT
  BULK COLLECT INTO AEW FROM A, W,
   (SELECT ROW AS COL,
           SUM(matmul(ERR, t(MAT))
             * reluderiv(ACT)) AS ERR
    FROM AEW WHERE LAYER=l
    GROUP BY ROW) AS E
  WHERE A.COL=W.ROW AND W.COL=E.COL
    AND A.LAYER=l-2 AND W.LAYER=l-1;
    AND A.ITER=i AND W.ITER=i;
end for

--Update the first set of weights (on the inputs)
SELECT i+1, 1, ROW, COL,
       MAT - matmul(t(ACT), ERR) * 0.00000001
BULK COLLECT INTO W
FROM AEW WHERE LAYER=1;
```

Figure 2: SQL code to implement the backward pass for iteration i of a feed-forward deep network with eight hidden layers.

ROW and COL again identify a particular matrix chunk. Given this, a fully model parallel implementation of the backward pass can be implemented using the SQL code in Figure 2.

crossentropyderiv() and reluderiv() are user-defined functions implementing the derivatives of cross-entropy and ReLU activation, respectively. The entire model parallel backward-pass code is around twenty lines long and could be generated by an auto-differentiation tool.

2.4 So, What's the Catch?

This code illustrates both the promise of expressing such tensor-based computations relationally, but also the pitfalls of asking the user to provide control flow. While the core computation is declarative, an imperative loop has been used to loop backward through the layers. The SQL programmer used a database table to pass state between iterations. In our example, this is done by utilizing the AEW table, which stores the error being back-propagated through each of the connections from layer $l+1$ to layer l in the network, for each of the data points in the current learning batch. If there are 100,000 neurons in two adjacent layers in a fully-connected network and 1,000 data points in a batch, then there are $(100{,}000)^2$ such connections for each of the 1,000 data points, or 10^{13} values stored in all. Using single-precision floating point value, a debilitating 40TB of data must be materialized.

Storing the set of per-connection errors is a very intuitive choice as a way to communicate among loops iterations, especially since the per-connection errors are subsequently aggregated in two ways (one to compute the new weights at a layer, and one to compute the new set of per-connection errors passed to the next layer). But forcing the system to materialize this table can result in a very inefficient computation. This *could* be implemented by pipelining the computation creating the new data for the AEW table directly into the two subsequent aggregations, but this possibility has been lost when the programmer asked that the new data be BULK COLLECTed into AEW.

Note that this is not merely a case of a poor choice on the part of the programmer. In order to write a loop, state has to be passed from one iteration to another, and it is this state that made it impossible for the system to realize an ideal implementation.

3. EXTENSIONS TO SQL

In this section, we consider a couple of extensions to SQL that make it possible for a programmer (either a human or a deep-learning toolchain) to declaratively specify recursive computations such as back-propagation, without control flow.

3.1 The Extensions

We introduce these SQL extensions in the context of a classic introductory programming problem: implementing Pascal's triangle, which recursively defines binomial coefficients. Specifically, the goal is to build a matrix such that the entry in row i and column j is $\binom{i}{j}$ (or i choose j). The triangle is defined recursively so that for any integers $i \geq 0$ and $j \in [1, i-1]$, $\binom{i}{j} = \binom{i-1}{j-1} + \binom{i-1}{j}$:

i \ j	0	1	2	3	4
0	1				
1	1	1			
2	1	2	1		
3	1	3	3	1	
4	1	4	6	4	1

Our extended SQL allows for multiple versions of a database table; versions are accessed via *array-style indices*. For example, we can define a database table storing the binomial coefficient $\binom{0}{0}$ as:

```
CREATE TABLE pascalsTri[0][0] (val) AS
  SELECT val FROM VALUES (1);
```

The table pascalsTri[0][0] can now be queried like any other database table, and various versions of the tables can be defined recursively. For example, we can define all of the cases where $j = i$ (the diagonal of the triangle) as:

```
CREATE TABLE pascalsTri[i:1...][i] (val) AS
  SELECT * FROM pascalsTri[i-1][i-1]
```

And all of the cases where $j = 0$ as:

```
CREATE TABLE pascalsTri[i:1...][0] (val) AS
  SELECT * FROM pascalsTri[i-1][0]
```

Finally, we can fill in the rest of the cells in the triangle via one more recursive relationship:

```
CREATE TABLE pascalsTri[i:2...][j:1...i-1](val) AS
  SELECT pt1.val + pt2.val AS val
  FROM pascalsTri[i-1][j-1] AS pt1,
       pascalsTri[i-1][j] AS pt2;
```

Note that this differs quite a bit from classical, recursive SQL, where the goal is typically to compute a fix-point of a set. Here, there is no fix-point computation. In fact, this particular recurrence defines an infinite number of versions of the pascalsTri table. Since there can be an infinite number of such tables, those tables are materialized on-demand. A programmer can issue the query:

```
SELECT * FROM pascalsTri[56][23]
```

In which case the system will unwind the recursion, writing the required computation as a single relational algebra statement. A programmer may ask questions about multiple versions of a table at the same time (without having each one be computed separately):

```
EXECUTE (
  FOR j IN 0...50:
    SELECT * FROM pascalsTri[50][j])
```

By definition, all of the queries/statements within an EXECUTE command are executed as part of the same query plan. Thus, this would be compiled into a single relational algebra statement that produces all 51 of the requested tables, under the constraint that each of those 51 tables must be materialized (without such a constraint, the resulting physical execution plan may pipeline one or more of those tables, so that they exist only ephemerally and cannot be returned as a query result). If a programmer wished to materialize all of these tables so that they could be used subsequently without re-computation, s/he could use:

```
EXECUTE (
  FOR j IN 0...50:
    MATERIALIZE pascalsTri[50][j])
```

which materializes the tables for later use. Finally, we introduce a multi-table UNION operator that merges multiple, recursively-defined tables. This makes it possible to define recursive relationships that span multiple tables. For example, a series of tables storing the various Fibonacci numbers (where $Fib(i) = Fib(i-1) + Fib(i-2)$ and $Fib(1) = Fib(2) = 1$) can be defined as:

```
CREATE TABLE Fibonacci[i:0...1] (val) AS
  SELECT * FROM VALUES (1)

CREATE TABLE Fibonacci[i:2...] (val) AS
  SELECT SUM (VAL) FROM UNION Fibonacci[i-2...i-1]
```

In general, UNION can be used to combine various subsets of recursively defined tables. For example, one could refer to UNION pascalsTri[i:0...50][0...i] which would flatten the first 51 rows of Pascal's triangle into a single multiset.

3.2 Learning Using Recursive SQL

With our SQL extensions, we can rewrite the aforementioned forward-backward passes to eliminate imperative control flow by declaratively expressing the various dependencies among the activations, weights, and errors.

Forward pass. The forward pass is concerned with computing the level of activation of the neurons at each layer. The activations of all neurons in layer j at learning iteration i are given in the table A[i][j]. Activations are computed using the weighted sum of the outputs of all of the neurons at the last level; the weighted sums input into layer j at learning iteration i is given in the table WI[i][j]. The corresponding SQL code is as follows. The forward pass begins by loading the first layer of activations with the input data:

```
CREATE TABLE A[i:0...][j:0](COL, ACT) AS
  SELECT DI.COL, DI.VAL
  FROM DATA_INPUT AS DI;
```

We then send the activation across the links in the network:

```
CREATE TABLE WI[i:0...][j:1...9](COL, VAL) AS
  SELECT W.COL, SUM(matmul(A.ACT, W.MAT))
  FROM W[i][j] AS w, A[i][j-1] AS A
  WHERE W.ROW = A.COL
  GROUP BY W.COL;
```

Those links are then used to compute activations:

```
CREATE TABLE A[i:0...][j:1...8](COL, ACT) AS
  SELECT WI.COL, relu(WI.VAL + B.VEC)
  FROM WI[i][j] AS WI, B[i][j] AS B
  WHERE WI.COL = B.COL;
```

And finally, at the top layer, the softmax function is used to perform the prediction:

```
CREATE TABLE A[i:0...][j:9](COL, ACT) AS
  SELECT WI.COL, softmax(WI.VAL + B.VEC)
  FROM WI[i][j] AS WI, B[i][j] AS B
  WHERE WI.COL = B.COL;
```

Backward pass. In the backward pass, the errors are pushed backward through the network. The error being pushed through layer j in learning iteration i are stored in the table E[i][j]. These errors are used to update all of the network's weights (the weights directly affecting layer j in learning iteration i are stored in W[i][j]) as well as biases (stored in B[i][j])).

We begin the SQL code for the backward pass with the initialization of the error:

```
CREATE TABLE E[i:0...][j:9](COL, ERR) AS
  SELECT A.COL, crossentropyderiv(A.ACT, DO.VAL)
  FROM A[i][j] AS A, DATA_OUTPUT AS DO;
```

At subsequent layers, the error is:

```
CREATE TABLE E[i:0...][j:1...8](COL, ERR) AS
  SELECT W.ROW, SUM(matmul(E.ERR, t(W.MAT))
                 * reluderiv(A.ACT))
  FROM A[i][j] AS A, E[i][j+1] AS E,
       W[i][j+1] AS W
  WHERE A.COL = W.ROW AND W.COL = E.COL
  GROUP BY W.ROW;
```

Now we use the error to update the weights:

```
CREATE TABLE W[i:1...][j:1...9](ROW, COL, MAT) AS
  SELECT W.ROW, W.COL,
         W.MAT - matmul(t(A.ACT), E.ERR) * 0.00000001
  FROM W[i-1][j] AS W, E[i-1][j] AS E,
       A[i-1][j-1] AS A
  WHERE A.COL = W.ROW AND W.COL = E.COL;
```

And the biases:

```
CREATE TABLE B[i:1...][j:1...9](COL, VEC) AS
  SELECT B.COL,
         B.VEC - reducebyrow(E.ERR) * 0.00000001
  FROM B[i-1][j] AS B, E[i-1][j] AS E
  WHERE B.COL = E.COL;
```

We now have a fully declarative implementation of neural network learning.

4. EXECUTING RECURSIVE PLANS

The recursive specifications of the last section address the problem of how to succinctly and declaratively specify complicated recursive computations. Yet the question remains: How can the very large and complex computations associated with such specifications be compiled and executed by an RDBMS without significant modification to the system?

4.1 Frame-Based Execution

Our possibility for compiling and executing computations written recursively in this fashion is to first compile the recursive computation into a single monolithic relational algebra computation, and then partition the computation into *frames*, or sub-plans. Those frames are then optimized and executed independently, with intermediate tables materialized to facilitate communication between frames.

Frame-based computation is attractive because if each frame is small enough that an existing query optimizer and execution engine can handle the frame, the RDBMS optimizer and engine need not be modified in any way. Further, this iterative execution results in an engine that resembles engines that perform re-optimization during runtime [12], in the sense that frames are optimized and executed only once all of their inputs have been materialized. Accurate statistics can be collected on those inputs—specifically, the number of distinct attribute values can be collected using an algorithm like Alon-Matias-Szegedy [1]—meaning that problems associated with errors propagating through a query plan can be avoided.

4.2 Heuristic vs. Full Unrolling

One could imagine two alternatives for implementing a frame-based strategy. The first is to rely on a heuristic, such as choosing the outer-most loop index, breaking the computation into frames using that index. However, there are several problems with this approach. First off, we are back to the problem described in Section 3.3, where we are choosing to materialize tables in an ad-hoc and potentially dangerous way (we may materialize a multi-terabyte table). Second, we cannot control the size of the frame. Too many operations can mean that the system is unable to optimize and execute the frame, while too few can mean a poor physical plan with too much materialized data. Third, if we allow the recursion to go up as well as down, or skip index values, this will not work.

Instead, we opt for an approach that performs a full unrolling of the recursive computation into a single, monolithic computation, which may in practice consist of hundreds of thousands of relational operations, and then define an optimization problem that attempts to split the computation into frames so as to minimize the likelihood of materializing a large number of tables.

4.3 Optimization Problem: Intuition

The cost incurred when utilizing frames is twofold. First, the use of frames restricts the ability of the system's logical and physical optimizer to find optimization opportunities. For example, if the logical plan $((R \bowtie S) \bowtie T)$ is optimal but the input plan $((R \bowtie T) \bowtie S)$ is cut into frames $f_1 = (R \bowtie T)$ and $f_2 = (f_1 \bowtie S)$ it is impossible to realize this optimal plan. In practice, we address this by placing a minimum size on frames as larger frames make it more likely that high-quality join orderings will still be present in the frame.

More significant is the requirement that the contents of already-executed frames be saved, so that later frames may utilize them. This can introduce significant I/O cost compared to a monolithic execution. Thus we we may attempt to cut into frames to minimize the number of bytes traveling over cut edges. Unfortunately, this is unreasonable as it is well-understood that estimation errors propagate through a plan; in the upper reaches of a huge plan, it is going to be impossible to estimate the number of bytes traveling over edges.

Instead, we find that spitting the plan into frames so as to reduce the number of *pipeline breakers* induced is a reasonable goal. A pipeline breaker occurs when the output of one operator must be materialized to disk or transferred over the network, as opposed to being directly communicated from operator to operator via CPU cache, or, in the worst case, via RAM. An induced pipeline breaker is one that would not have been present an optimal physical plan, but was forced by the cut.

4.4 Quadratic Assignment Formulation

Given a query plan, it is unclear whether a cut that separates two operators into different frames will induce a pipeline breaker. We model this uncertainty using probability, and seek to minimize the expected number of pipeline breakers induced by the set of chosen frames.

This is "probability" in the Bayesian rather than frequentist sense, in that it represents a level or certainty or belief in the pipelineability of various operators. For the ith and jth operators in the query plan, let N_{ij} be a random variable that takes the value 1 if operator i is pipelined into operator j were the entire plan optimized and executed as a unit, and 0 otherwise.

Let the query plan to be cut into frames be represented as a directed graph having n vertices, represented as a binary matrix $\mathbf{E}$, where e_{ij} is one (that is, there is an edge from vertex i to vertex j) if the output of operator i is directly consumed by operator j. e_{ij} is zero otherwise. We would like to split the graph into m frames. We define the *split* of a query plan to be a matrix $\mathbf{X} = (x_{ij})_{n\times n}$, where each row would be one frame so that $x_{ij} = 1$ if operator i is in a different frame from operator j (that is, they have been cut apart) and 0 otherwise. Given this, the goal is to minimize:

$$cost(\mathbf{X}) = E\left[\sum_{i=1}^{n}\sum_{j=1}^{n} e_{ij}x_{ij}N_{ij}\right] = \sum_{i=1}^{n}\sum_{j=1}^{n} e_{ij}x_{ij}E\left[N_{ij}\right]$$

This computes the expected number of pipeline breakers induced, as for us to induce a new pipeline breaker via the cut, (a) operator j must consume the output from operator i, (b) operator i and j must be separated by the cut, and (c) operator i should have been pipelined into operator j in the optimal execution.

We can re-write the objective function by instead letting the matrix $\mathbf{X} = (x_{ij})_{n\times m}$ be an *assignment matrix*, where $\sum_i x_{ij} = 1$, and each x_{ij} is either one ore zero. Then, x_{ij} is one if operator i is put into frame j and we have:

$$cost(\mathbf{X}) = \left(\sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{a=1}^{m}\sum_{b=1}^{m} e_{ij}x_{ia}x_{jb}E\left[N_{ij}\right]\right) - \left(\sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{a=1}^{m} e_{ij}x_{ia}x_{ja}E\left[N_{ij}\right]\right)$$

Letting $c_{ijab} = e_{ij}E\left[N_{ij}\right] - \delta_{ab}e_{ij}E\left[N_{ij}\right] = e_{ij}E\left[N_{ij}\right](1-\delta_{ab})$ where δ_{ab} is the Kronecker delta function, we then have:

$$cost(\mathbf{X}) = \sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{a=1}^{m}\sum_{b=1}^{m} c_{ijab}x_{ia}x_{jb}$$

The trivial solution to choosing $\mathbf{X}$ to minimize this cost function is to put all or most operators in the same frame, but that would result in a query plan that is not split in a meaningful way. Therefore we need to add a constraint on the upper bound of operators in each frame: $min \leq \sum_j x_{ij} \leq max$ for some maximum frame size.

The resulting optimization problem is not novel: it is an instance of the problem popularly known as the *generalized quadratic assignment problem*, or GQAP [14], where the goal is to map tasks or machinery (in this case, the various operations we are executing) into locations or facilities (in this case, the various frames). GQAP generalizes the classical quadratic assignment problem by allowing multiple tasks or pieces of machinery to be mapped into the same location or facility (in the classical formulation, only one task is allowed per facility). Unfortunately, both GQAP and classical quadratic assignment are NP-hard, and inapproximable.

In our instance of the problem, we actually have one additional constraint that is not expressible within the standard GQAP framework. A simple minimization of the objective function could result in a sequence of frames that may not be executable because they contain circular dependencies. In order to ensure that we have no circular dependencies, we have to make the intermediate value that a frame uses available before it is executed. To do this, we take the natural ordering of the frames to be meaningful, in the sense that frame a is executed before frame b when $a < b$, and for each edge e_{ij} in the computational graph, we introduce the constraint that for a, b where $x_{ia} = 1$ and $x_{jb} = 1$, it must be the case that $a \leq b$.

4.5 Cost Model

So far, we have not discussed the precise nature of the various N_{ij} variables that control whether the output of operator i is pipelined into operator j in a single, uncut, optimized and executed version of the computation. Specifically, we need to compute the value of $E[N_{ij}]$ required by our GQAP instance. Since each N_{ij} is a binary variable, $E[N_{ij}]$ is simply the probability that N_{ij} evaluates to one. Let p_{ij} denote this probability. In keeping with our view, we define the various p_{ij} values as follows:

(1) If the output of operator i has one single consumer (operator j) and operator j is a selection or an aggregation, then p_{ij} is 1. The reason for this is that in the system we are building on (SimSQL [2]), it is always possible to pipeline into a selection or an aggregation. Selections are always pipelineable, and in SimSQL, if operator j is an aggregation, then a corresponding pre-aggregation will be added to the end of the pipeline executing operation j. This pre-aggregation maintains a hash table for each group encountered in the aggregation, and as new data are encountered, statistics for each data object are added to the corresponding group. As long as the number of groups is small and the summary statistics compact, this can radically reduce the amount of data that needs to be shuffled to implement the aggregation.

(2) If the output of operator i has one single consumer (operator j) but operator j is not a selection or an aggregation, then p_{ij} is estimated using past workloads. That is, based off of workload history, we compute the fraction of the time that operator i's type of operation is pipelined into the type of operator j's operation, and use that for p_{ij}.

(3) In SimSQL, if operator i has multiple consumers, then the output of operator i can be pipelined into only one of them (the output will be saved to disk and then the other operators will be executed subsequently, reading the saved output). Hence, if there are k consumers of operator i, and operator j is a selection or an aggregation, then $p_{ij} = \frac{1}{k}$. Otherwise, if, according to workload history, the traction of the time that operator i's type of operation is pipelined into the type of operator j's operation is f, then $p_{ij} = \frac{f}{k}$.

5. EXPERIMENTS

5.1 Overview

In this section, we detail a set of experiments aimed at answering the following questions:

Can the ideas described in this paper be used to re-purpose an RDBMS so that it can be used to implement scalable, performant, model parallel ML computations?

We implement the ideas in this paper on top of SimSQL, a research-prototype, distributed database system [2]. SimSQL has a cost-based optimizer, an assortment of implementations of the standard relational operations, the ability to pipeline those operations and make use of "interesting" physical data organizations. It also has native matrix and vector support [16].

Scope of Evaluation. We stress that this is not a "which system is faster?" comparison. SimSQL is implemented in Java and runs on top of Hadoop MapReduce, with the high latency that implies. Hence a platform such as SimSQL is likely to be considerably faster than SimSQL, at least for learning smaller models (when SimSQL's high fixed costs will dominate).

Rather than determining which system is faster, the specific goal is to study whether an RDBMS-based, model-parallel learner is a viable alternative to a system such as TensorFlow, and whether it has any obvious advantages.

Experimental Details. In all of our experiments, all implementations run the same algorithms over the same data. Thus, a configuration that runs each iteration 50% faster than another configuration will reach a given target loss value (or log-likelihood) 50% faster. Hence, rather than reporting loss values (or log-likelihoods) we report per-iteration running times.

All implementations are fully synchronous, for an apples-to-apples comparison. We choose synchronous learning as there is strong evidence that synchronous learning for large, dense problems is the most efficient choice [3, 9].

In the first set of FFNN experiments, EC2 `r5d.2xlarge` CPU machines with 8 cores and 64GB of RAM were used. In the second set, at various cost levels, we chose sets of machines to achieve the best performance. For TensorFlow, this was achieved using GPU machines; for SimSQL, both CPU and GPU machines achieved around the same performance.

We use the data parallel, synchronous, feed-forward network implementation that ships with TensorFlow as a comparison with the FFNN implementation described in this paper. We use a Wikipedia dump of 4.86 million documents as the input to the feed-forward learner. The goal is to learn how to predict the year of the last edit to the article. There are 17 labels in total. We process the Wikipedia dump, representing each document as a 60,000-dimensional feature vector. In most experiments, we use a size 10,000 batch.

5.2 Results

To examine the necessity of actually using a frame-based execution, we use ten machines to perform FFNN learning on a relatively small learning task (10,000 hidden neurons, batch size 100). We unroll 60 iterations of the learning, and compare the per-iteration running time using the full cutting algorithm along with the cost model of Section 6.3 with a monolithic execution of the entire, unrolled plan. The resulting graph has 12,888 relational operators. The monolithic execution failed during the second iteration. The per-iteration running time of the frame-based execution is compared with monolithic execution in Figure 3.

We evaluate both the RDBMS and TensorFlow with a variety of cluster sizes (five, ten, and twenty machines) and a wide variety of hidden layer sizes—up to 160,000 neurons. onnecting two such layers requires a matrix with 26 billion entries (102 GB). Per-iteration execution times are given in Figure 4. "Fail" means that the system crashed.

In addition, we ran a set of experiments where we attempted to achieve the best performance at a \$3-per-hour, \$7-per-hour, and \$15-per-hour price point using Amazon AWS. For TensorFlow, at \$3, this was one `p3.2xlarge` GPU machine and a `r5.4xlarge` CPU machine; at \$7, it was two `p3.2xlarge` GPU machines and two `r5.4xlarge` CPU machines, and at \$15, it was four `p3.2xlarge` GPU machines and four `r5.4xlarge` CPU ma-

Graph Type	FFNN per-iteration time
Whole Graph	05:53:29
Frame-Based	00:12:53

Figure 3: Frame-based vs. monolithic execution.

FFNN		
Hidden Layer Neurons	RDBMS	TensorFlow
Cluster with 5 workers		
10000	05:39	01:36
20000	05:46	03:38
40000	08:30	09:02
80000	24:52	Fail
160000	Fail	Fail
Cluster with 10 workers		
10000	04:53	00:54
20000	05:32	02:00
40000	07:41	04:59
80000	17:46	Fail
160000	44:21	Fail
Cluster with 20 workers		
10000	04:08	00:32
20000	05:40	01:12
40000	06:13	02:56
80000	12:55	Fail
160000	25:00	Fail

Figure 4: Average iteration time for FFNN learning, using various CPU cluster and hidden layer sizes.

chines. SimSQL did about the same using one, two or four `c5d.18 xlarge` CPU machines (at \$3, \$7, and \$15, respectively) as it did using two, five or ten `c5d.18xlarge` GPU machines. Per-iteration execution times are given in Figure 5.

5.3 Discussion

SimSQL was unable to handle the 12,888 operators in the FFNN plan, resulting in a running time that was around 100× longer than frame-based execution (see Figure 3).

On the CPU clusters (Figure 4), the RDBMS was slower than TensorFlow in most cases, but it scaled well, whereas TensorFlow crashed (due to memory problems) on a problem size of larger than 40,000 hidden neurons.

Micro-benchmarks showed that for the 40,000 hidden neuron problem, all of the required matrix operations required for an iteration of FFNN learning took 6 minutes, 17 seconds on a single machine. Assuming a perfect speedup, on five machines, learning should take just 1:15 per iteration. However, the RDBMS took 8:30, and TensorFlow took 9:30. This shows that both systems incur significant overhead, at least at such a large model size. SimSQL, in particular, requires a total of 61 seconds per FFNN iteration just starting up and tearing down Hadoop jobs. As the system uses Hadoop, each intermediate result that cannot be pipelined must be written to disk, causing a significant amount of I/O. A faster database could likely lower this overhead significantly.

On a GPU (Figure 5) TensorFlow was very fast, but could not scale past 10,000 neurons. The problem is that when using a GPU, all data in the compute graph must fit on the GPU; TensorFlow is not designed to use CPU RAM as a buffer for GPU memory. The result is that past 10,000 neurons (where one weight matrix is 4.8GB in size) GPU memory is inadequate and the system fails.

Our GPU support in SimSQL did not provide much benefit, for a few reasons. First, the AWS GPU machines do not have attached

FFNN			
Hidden Layer Size	RDBMS (CPU)	RDBMS (GPU)	TensorFlow (GPU)
\$3 per hour budget			
10000	04:50	06:25	00:24
20000	07:07	07:12	Fail
40000	11:52	11:48	Fail
80000	16:30	Fail	Fail
160000	Fail	Fail	Fail
\$7 per hour budget			
10000	04:53	04:58	00:15
20000	05:54	06:08	Fail
40000	09:32	08:26	Fail
80000	12:03	17:50	Fail
160000	Fail	Fail	Fail
\$15 per hour budget			
10000	05:12	5:00	00:12
20000	05:36	06:30	Fail
40000	09:08	08:39	Fail
80000	12:24	12:20	Fail
160000	39:40	Fail	Fail

Figure 5: Average iteration time for FFNN learning, maximizing performance at a specific dollar cost.

storage, which means that moving to GPU machines meant that all of the disk read/writes incurred by Hadoop had to happen over network attached storage (compare with the CPU hardware, which had a fast, attached solid-state drive). Second, as discussed above, SimSQL's overhead above and beyond pure CPU time for matrix operations is high enough that reducing the matrix time further using a GPU was ineffective.

6. BACKGROUND AND RELATED WORK

During learning, we are given a data set $\mathbf{T}$ with elements $\mathbf{t}_j$. The goal is to learn a d-dimensional vector ($d \geq 1$) of model parameters $\Theta = (\Theta^{(1)}, \Theta^{(2)}, \ldots, \Theta^{(d)})$ that minimize a loss function of the form $\sum_j L(\mathbf{t}_j|\Theta)$. To this end, learning algorithms such as gradient descent perform a simple update repeatedly until convergence:

$$\Theta_{i+1} \leftarrow \Theta_i - F(\Theta_i, \mathbf{T})$$

If it is possible to store Θ_i in the RAM of each machine, decomposable learning algorithms can be made *data parallel*. One can broadcast Θ_i to each site, and then compute $F(\Theta_i, \mathbf{t}_j)$ for data $\mathbf{t}_j$ stored locally. All of these values are then aggregated using standard, distributed aggregation techniques.

However, data parallelism of this form is often ineffective. Let $\mathbf{T}_i$ be a small sample of $\mathbf{T}$ selected during epoch i. Since for decomposable algorithms, $F(\Theta_i, \mathbf{T}) \approx \frac{|\mathbf{T}|}{|\mathbf{T}_i|} F(\Theta_i, \mathbf{T}_i)$, in practice only a small subsample of the data are used during each epoch (for example, in the case of gradient descent, *mini-batch gradient descent* [18] is typically used). Adding more machines can either distribute this sample so that each machine gets a tiny amount of data (which is typically not helpful because for very small data sizes, the fixed costs associated with broadcasting Θ_i dominate) or else use a larger sample. This is also not helpful because the estimate to $F(\Theta_i, \mathbf{T})$ with a relatively small sample is already accurate enough. The largest batches advocated in the literature consist of around 10,000 samples [9].

One idea to overcome this is to use *asynchronous data parallelism* [17], where recursion of the form $\Theta_{i+1} \leftarrow \Theta_i - F(\Theta_i, \mathbf{T})$ is no longer used. Rather, each site j is given a small sample

$\mathbf{T}_j$ of $\mathbf{T}$; it requests the value Θ_{cur}, computes $\Theta_{new} \leftarrow \Theta_{cur} - F(\Theta_{cur}, \mathbf{T}_j)$ and registers Θ_{new} at a parameter server. All requests for Θ_{cur} happen to obtain whatever the last value written was, leading to stochastic behavior. The problem is that data parallelism of this form can be ineffective for large computations as most of the computation is done using stale data [3]. An alternative is *model parallelism*. In model parallelism, the idea is to stage $F(\Theta_i, \mathbf{T})$ (or $F(\Theta_i, \mathbf{T}_i)$) as a distributed computation without assuming that each site has access to (or stores) all of Θ_i (or $\mathbf{T}_i$).

The parameter server architecture [20, 15] was proposed to provide scalable, parallel training for machine learning models. It is favored by most existing Big Data ML systems (such as TensorFlow [6, 7] and Petuum [23]). A parameter server consists of two components: a parameter server (or key-value store) and a set of workers who repeatedly access and update the model parameters. Model parallelism is enabled in TensorFlow by distributing the nodes of a neural network across different machines. Although it provides some functions (e.g., `tf.nn.embedding_lookup`) that allow parallel model updates, support for more complex parallel model updates is limited. Petuum [23] considers to speed up distributed training, using ideas such as sending weights as soon as they are updated during backpropagation. MXNet [4] is anther system that employs a parameter server to train neural networks. MXNet claims to support model parallelism. However, its model parallelism support is similar to TensorFlow. Complex, model-parallel computations require using low-level APIs and manual management of the computations and communications.

There are several other systems providing model parallelism [13]. AMPNet [8] adds control flow to the execution graph, and supports dynamic control flow by introducing a well-defined intermediate representation. This framework proves to be efficacy for asynchronous model-parallel training by the experiments. Coates et al. [5] built a distributed system on a cluster of GPUs based on the COTS HPC technology. This system achieved model parallelism by carefully assigning the partial computations of the whole model to each GPU, and utilized MPI for the communication.

7. CONCLUSIONS

We have argued that a parallel/distributed RDBMS has promise as a backend for implementing and executing large scale ML computations. We have considered unrolling recursive computations into a monolithic compute plan, which is broken into frames that are optimized and executed independently. We have expressed the frame partitioning problem as an instance of the GQAP. When implemented on top of an RDBMS, these ideas result in ML computations that are model parallel—that is, able to handle large and complex models that need to be distributed across compute units.

8. REFERENCES

[1] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. In *STOC*, pages 20–29. ACM, 1996.

[2] Z. Cai, Z. Vagena, L. Perez, S. Arumugam, P. J. Haas, and C. Jermaine. Simulation of database-valued markov chains using simsql. In *SIGMOD 2013*, pages 637–648. ACM, 2013.

[3] J. Chen, X. Pan, R. Monga, S. Bengio, and R. Jozefowicz. Revisiting distributed synchronous sgd. *arXiv preprint arXiv:1604.00981*, 2016.

[4] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. *arXiv preprint arXiv:1512.01274*, 2015.

[5] A. Coates, B. Huval, T. Wang, D. J. Wu, A. Y. Ng, and B. Catanzaro. Deep learning with cots hpc systems. In *ICML 2013*, ICML'13, pages III–1337–III–1345. JMLR.org, 2013.

[6] M. A. et. al. Tensorflow: A system for large-scale machine learning. In *OSDI 16*, pages 265–283, GA, 2016. USENIX Association.

[7] M. A. et al. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. *arXiv preprint arXiv:1603.04467*, 2016.

[8] A. L. Gaunt, M. A. Johnson, M. Riechert, D. Tarlow, R. Tomioka, D. Vytiniotis, and S. Webster. AMPNet: Asynchronous Model-Parallel Training for Dynamic Neural Networks. *arXiv preprint arXiv:1705.09786*, 2017.

[9] P. Goyal, P. Dollár, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He. Accurate, large minibatch sgd: training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

[10] W. D. Hillis and G. L. Steele, Jr. Data parallel algorithms. *Commun. ACM*, 29(12):1170–1183, Dec. 1986.

[11] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[12] N. Kabra and D. J. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In *ACM SIGMOD Record*, volume 27, pages 106–117. ACM, 1998.

[13] A. Krizhevsky. One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997*, 2014.

[14] C.-G. Lee and Z. Ma. The generalized quadratic assignment problem. 01 2004.

[15] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su. In *OSDI*, pages 583–598, Berkeley, CA, USA.

[16] S. Luo, Z. J. Gao, M. Gubanov, L. L. Perez, and C. Jermaine. Scalable linear algebra on a relational database system. In *ICDE 2017*, pages 523–534. IEEE, 2017.

[17] B. Recht, C. Re, S. Wright, and F. Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *NIPS*, pages 693–701, 2011.

[18] S. Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[19] N. Shazeer, A. Mirhoseini, K. Maziarz, A. Davis, Q. V. Le, G. E. Hinton, and J. Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *CoRR*, abs/1701.06538, 2017.

[20] A. Smola and S. Narayanamurthy. An architecture for parallel topic models. *Proc. VLDB Endow.*, 3(1-2):703–710, Sept. 2010.

[21] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[22] E. W. Weisstein. Einstein summation. 2014.

[23] E. P. Xing, Q. Ho, W. Dai, J. K. Kim, J. Wei, S. Lee, X. Zheng, P. Xie, A. Kumar, and Y. Yu. Petuum: A new platform for distributed machine learning on big data. *IEEE Transactions on Big Data*, 1(2):49–67, June 2015.

Technical Perspective: Efficient Logspace Classes for Enumeration, Counting, and Uniform Generation

Reinhard Pichler
TU Wien, Austria

Traditionally, by *query answering* we mean the problem of finding *all* answers to a given query over a given database. But what happens if the number of answers is prohibitively big – which may easily occur in a Big Data context? In such situations, it seems preferable to have a mechanism that produces one answer after the other with certain guarantees on the time between any two outputs and to let the user decide when to stop. This leads us to the *enumeration problem*, which has received a lot of interest recently [1]. However, in order for the user to get a "realistic" picture of the entirety of answers, two crucial questions arise: first, how big is the portion of output answers compared with the total number of answers? And second, do the output answers reflect the variety of the complete set of answers? The first question refers to the *counting* problem, where we are interested in the total number of answers. The second question leads us to the problem of *uniform generation*, where we request that the answers be uniformly generated and thus form an unbiased sample of the complete set of answers.

Decades of research have been devoted to the analysis of the *complexity* of query answering of all kinds of query languages and, in particular, to the identification of scenarios (by imposing restrictions on the queries and/or databases) which allow for *efficient* query answering. Such complexity analyses usually consider a suitable decision problem such as checking if a given candidate is indeed an answer to a given query over the given database. As we shift our focus to the triad of enumeration, counting, and uniform generation, the notions of *complexity* and of *efficiency* have to be reconsidered. Moreover, in order to get an understanding of the complexity of a concrete query language for a concrete type of databases, it seems unavoidable to analyse the three problems separately . . .

. . . unless we have a general (and, ideally, simple) framework that allows us to detect at once if all three problems are efficiently solvable. Establishing such a framework is the very goal of the paper *Efficient Logspace Classes for Enumeration, Counting, and Uniform Generation* by Marcelo Arenas, Luis Alberto Croquevielle, Rajesh Jayaram, and Cristian Riveros. The resulting framework is very general and by no means restricted to query answering. It follows the classical formalism of [2], which represents a *problem* as a binary relation R consisting of pairs (x, y), where x is (an encoding of) the *input* and y is (an encoding of) a *solution*. In our case of query answering, the input consists of a database and a query and each answer corresponds to a solution. The authors define classes of such binary relations by means of Turing machines. As a good compromise between expressive power (i.e., capturing an interesting class of relations) and efficiency (i.e., making sure that enumeration, counting, and uniform generation all have acceptable complexity), only non-deterministic log-space Turing machines M are considered, i.e., when run on some input x, machine M produces as output in its accepting computations precisely the solutions y with $(x, y) \in R$. The resulting class of relations is referred to as RelationNL. This compromise between expressive power and efficiency can be shifted a bit more towards efficiency (at the expense of less expressive power) by requiring the Turing machine to be *unambiguous*, i.e., any two different runs of M on input x ending in an accepting state produce distinct outputs. The resulting class of relations is referred to as RelationUL.

The main technical results of the paper are upper bounds on the complexity of enumeration, counting, and uniform generation if a relation is in RelationNL or in RelationUL, respectively. For instance, if R is in RelationUL, then the enumeration problem can be solved with *constant delay* (i.e., after a polynomial-time preprocessing phase, the time between any two successive outputs is independent of the size of the input), the counting problem is polynomial-time solvable and, finally, uniform generation is feasible by a polynomial-time randomized algorithm. Slightly less favorable upper bounds hold for relations in RelationNL. The paper also shows that several interesting problems from various domains (including information extraction and graph databases) indeed fall in one of these two relation-classes.

The paper represents an important first step that opens up many research opportunities such as proving the membership of various problems in RelationNL or RelationUL, searching for further optimizations of the enumeration, counting and uniform generation algorithms for problems in these classes, and also searching for interesting extensions and restrictions of these classes.

1. REFERENCES

[1] E. Boros, B. Kimelfeld, R. Pichler, and N. Schweikardt. Enumeration in data management (dagstuhl seminar 19211). *Dagstuhl Reports*, 9(5):89–109, 2019.

[2] M. Jerrum, L. G. Valiant, and V. V. Vazirani. Random generation of combinatorial structures from a uniform distribution. *Theor. Comput. Sci.*, 43:169–188, 1986.

Efficient Logspace Classes for Enumeration, Counting, and Uniform Generation

Marcelo Arenas
PUC & IMFD Chile
marenas@ing.puc.cl

Luis Alberto Croquevielle
PUC & IMFD Chile
lacroquevielle@uc.cl

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

Cristian Riveros
PUC & IMFD Chile
cristian.riveros@uc.cl

ABSTRACT

We study two simple yet general complexity classes, which provide a unifying framework for efficient query evaluation in areas like graph databases and information extraction, among others. We investigate the complexity of three fundamental algorithmic problems for these classes: enumeration, counting and uniform generation of solutions, and show that they have several desirable properties in this respect.

Both complexity classes are defined in terms of non deterministic logarithmic-space transducers (NL transducers). For the first class, we consider the case of unambiguous NL transducers, and we prove constant delay enumeration, and both counting and uniform generation of solutions in polynomial time. For the second class, we consider unrestricted NL transducers, and we obtain polynomial delay enumeration, approximate counting in polynomial time, and polynomial-time randomized algorithms for uniform generation. More specifically, we show that each problem in this second class admits a fully polynomial-time randomized approximation scheme (FPRAS) and a polynomial-time Las Vegas algorithm (with preprocessing) for uniform generation. Remarkably, the key idea to prove these results is to show that the fundamental problem #NFA admits an FPRAS, where #NFA is the problem of counting the number of strings of length n (given in unary) accepted by a non-deterministic finite automaton (NFA). While this problem is known to be #P-complete and, more precisely, SpanL-complete, it was open whether this problem admits an FPRAS. In this work, we solve this open problem, and obtain as a welcome corollary that every function in SpanL admits an FPRAS.

1. INTRODUCTION

Arguably, query answering is the most fundamental problem in databases. In this respect, developing efficient query answering algorithms, as well as understanding when this cannot be done, is of paramount importance in the field. In the most classical view of this problem, one is interested in computing all the answers, or solutions, to a query. To present such a view, consider a running example from the area of graph databases [8].

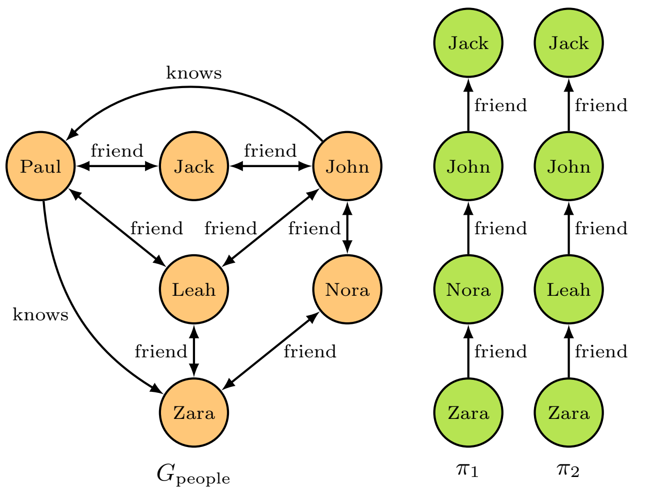


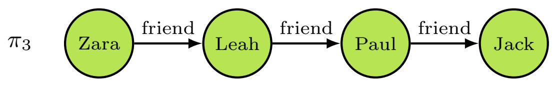
Figure 1: A graph database G_{people}, and two paths π_1, π_2 of friends in G_{people} from Zara to Jack.

Given a set Δ of labels, one can model a graph database G as a pair (V, E) where V is a set of vertices and $E \subseteq V \times \Delta \times V$ is a set of labeled edges. For example, G_{people} in Figure 1 is a graph database storing information about people and their relationships; in particular, the set of labels for G_{people} is $\{\text{friend}, \text{knows}\}$, so that a triple (a, friend, b) indicates that a and b are friends, while a triple (a, knows, b) indicates that a knows b. Path queries are a fundamental way to retrieve information from graph databases [8, 18]. In its simplest form, a path query Q over a graph database $G = (V, E)$ is a triple (a, r, b), where $a, b \in V$ and r is a regular expression over the set Δ of edge labels for G. An answer to Q over G is a path from a to b whose labels conform to r. Formally, such a path is a sequence $\pi = v_0, p_1, v_1, p_2, \ldots, p_n, v_n$ of vertices and labels such that $(v_i, p_{i+1}, v_{i+1}) \in E$, $a = v_0$ and $b = v_n$. Moreover, π is said to conform to r if the string $p_1 p_2 \cdots p_n$ is in the regular language defined by r. For example, $Q_1 = (\text{Zara}, \text{friend}^*, \text{Jack})$ is a path query over the graph database G_{people} in Figure 1, for which two answers are the paths π_1, π_2 shown in this figure. Thus, an answer to Q_1 over G_{people}

This is a minor revision of the paper entitled "Efficient Logspace Classes for Enumeration, Counting, and Uniform Generation", published in PODS'19, ISBN978-1-4503-6227-6/19/06, June 30-July 5, 2019, Amsterdam, Netherlands. DOI: http://dx.doi.org/10.1145/3294052.3319704 is a path of friends from Zara to Jack. The set of answers of a path query Q over a graph database G is denoted by $Q(G)$. Clearly, $Q(G)$ can be an infinite set, as paths can contain cycles, so there can be an infinite number of them in a graph. For this reason, the length of the paths to be retrieved must also be specified when posing a path query; the length of a path $\pi = v_0, p_1, \ldots, p_n, v_n$, denoted by $|\pi|$, is defined as n. Hence, in the most classical view of the query answering problem in graph databases, the input is a triple (G, Q, n) with G a graph database, Q a path query over G and n a natural number. The task then is to compute *all* paths π such that $\pi \in Q(G)$ and $|\pi| = n$. For example, assuming that Q_1 = (Zara, friend*, Jack), the paths π_1, π_2 in Figure 1 belong to $Q_1(G_{\text{people}})$ when $n = 3$.

As the quantity of data becomes enormously large, the number of answers to a query could also be enormous, so computing the complete set of solutions can be prohibitively expensive. In our running example, just think about computing all the paths from a source to a target node over a large graph, this set of answers can be huge and infeasible to produce in practice [9, 28]. To overcome this limitation, the idea of enumerating the answers to a query with a *small delay* has recently attracted a lot of attention [33, 29]. More specifically, the idea is to divide the computation of the answers into two phases. In a *preprocessing* phase, some data structures are constructed to accelerate the process of computing answers. Then in an *enumeration* phase, the answers are enumerated with a small delay between them. Considering again G_{people} and path query Q_1 = (Zara, friend*, Jack), such an algorithm has a preprocessing phase that allows it to return a first path in $Q_1(G_{\text{people}})$ in polynomial time, say π_1, and then to return one by one the answers in $Q_1(G_{\text{people}})$ taking polynomial time between any two consecutive outputs, say taking polynomial time to return π_2 after π_1. In the case of constant delay enumeration algorithms, the preprocessing phase should take polynomial time, while the time between consecutive answers should be constant. Such algorithms allow users to retrieve a fixed number of answers very efficiently, which can give them a lot of information about the solutions to a query. In fact, the same holds if users need a polynomial number of answers.

Unfortunately, because of the data structures used in the preprocessing phase, these enumeration algorithms usually return answers that are similar to each other [12, 33, 17]. In our running example, an enumeration algorithm for the query Q_1 = (Zara, friend*, Jack) can return as the first two answers the paths π_1 and π_2 shown in Figure 1, which are similar to each other as they only differ on the second node: Nora and Leah. In this respect, other approaches can be used to return some solutions efficiently but improving the variety. For instance, if we are going to generate two answers to Q_1 over G_{people}, instead of producing paths π_1 and π_2 in Figure 1, it would be desirable to improve the variety by producing π_1 and the following path:



The possibility of generating an answer uniformly, at random, is a desirable condition to improve the variety, if it can be done efficiently. Notice that the uniform generation of answers is also important for other query evaluation tasks like approximate query answering, and estimating aggregates and parameters for query optimization [14, 3, 36]. Moreover, the possibility of returning varied solutions has been identified as an important feature not only in databases, but also for algorithms that retrieve information in a broader sense [2, 1].

Efficient algorithms for enumeration or uniform generation are powerful tools to help in the process of understanding the answers to a query. But how can we know how long these algorithms should run, and how complete the set of computed answers is? A third tool that is needed then is an efficient algorithm for computing, or estimating, the number of solutions to a query. For example, for the query Q_1 = (Zara, friend*, Jack) over the graph database G_{people} in Figure 1, we have that $Q_1(G_{\text{people}})$ contains three paths π such that $|\pi| = 3$. Hence, if we have an efficient algorithm to compute the number of paths in $Q_1(G_{\text{people}})$ of length 3, then we know that there are no more answers to produce after generating paths π_1 and π_2 in Figure 1, and the previous path π_3. Similar than for enumeration and uniform generation, counting the number of solutions has other applications in query evaluation like computing the size of intermediate results, computing histograms, among others [20, 23].

Taken together, enumeration, counting, and uniform generation techniques form a powerful attacking trident when confronting to the query answering problem in our running example and, more generally, in any database system. The goal of this work is to find efficient algorithms for these problems but following a principled approach, instead of focusing on them in isolation and for some specific domains. More precisely, we follow the guidance of [24], which urges the use of relations to formalize the notion of solution to a given input of a problem, so that enumeration, counting, and uniform generation appear as particular problems in this formalization. In Section 2, we present this framework together with the formal notions of efficiency that we pursue for these three problems. The next step then is to provide a simple way to identify relations that have good properties in terms of these three tasks. For this, we use the concept of non-deterministic logspace transducers to provide two classes of relations, called RelationNL and RelationUL, and show that the aforementioned three problems admit efficient algorithms when restricted to these classes of relations. RelationNL and RelationUL are formally defined in Section 3, where our main results are also formally stated. Interestingly, one can show that several problems in data management are in one of these two classes. More specifically, by establishing membership in one of these two classes of relations, we show in Section 4 that problems about information extraction and binary decision diagrams, as well as our running example, admit efficient algorithms for enumeration, counting, and uniform generation.

It is important to mention that the main technical result of this work is to prove that each problem in RelationNL admits a fully polynomial-time randomized approximation scheme (FPRAS) [24] and a polynomial-time Las Vegas algorithm (with preprocessing) for uniform generation. The key idea to prove these results is to show that the fundamental problem #NFA admits an FPRAS, where #NFA is the problem of counting the number of strings of length n accepted by a non-deterministic finite automaton (NFA). While this problem is known to be #P-complete and, more precisely, SpanL-complete [5], it was open whether it admits an FPRAS, and only quasi polynomial time randomized approximation schemes were known for it [26, 21]. In this work, we solve this open problem, and obtain as a welcome corollary that every function in SpanL admits an FPRAS. Thus, to the best of our knowledge, we obtain the first complexity class with a simple and robust definition based on Turing Machines, that contains #P-complete problems and where each problem admits an FPRAS.

2. A UNIFYING FRAMEWORK BASED ON RELATIONS

As mentioned in the introduction, we follow a principled approach to study the problems of enumerating, counting and uniformly generating the answers to a query. We begin by following the guidance of [24], which urges the use of relations to formalize the notion of solution to a given input of a problem. Formally, if Σ denotes a finite alphabet, then we represent a problem as a relation $R \subseteq \Sigma^* \times \Sigma^*$, where, as usual, Σ^* denotes the set of all strings over Σ. For every pair $(x, y) \in R$, we interpret x as being the encoding of an input to the problem, and y as being the encoding of a solution or witness to that input. For each $x \in \Sigma^*$, we define the set $W_R(x) = \{y \in \Sigma^* \mid (x, y) \in R\}$, and call it the witness set for x. Also, if $y \in W_R(x)$, we call y a witness or a solution to x. For instance, the query answering problem from our running example in Figure 1 can be represented as the following relation:

$$\begin{aligned}\text{EVAL-PQ} = \{\, ((G, Q, n),\ \pi) \mid\ & G \text{ is a graph database,} \\ & Q \text{ is a path query, } n \in \mathbb{N},\ \pi \in Q(G), \text{ and } |\pi| = n \,\}, \qquad (\dagger)\end{aligned}$$

that is, the input to the query answering problem is the triple (G, Q, n), and a solution for (G, Q, n) is a path π such that $\pi \in Q(G)$ and the length of π is n. Thus, the following is the set of solutions for (G, Q, n):

$$W_{\text{EVAL-PQ}}((G, Q, n)) = \{\, \pi \mid ((G, Q, n),\ \pi) \in \text{EVAL-PQ}\,\}.$$

This is a very general framework where any relation between input and solutions can be encoded, so we restrict to *p*-relations [24]. Formally, a relation $R \subseteq \Sigma^* \times \Sigma^*$ is a *p*-relation if (1) there exists a polynomial q such that $(x, y) \in R$ implies that $|y| = q(|x|)$ and (2) there exists a deterministic Turing Machine that receives as input $(x, y) \in \Sigma^* \times \Sigma^*$, runs in polynomial time and accepts if, and only if, $(x, y) \in R$. One can easily check that EVAL-PQ is a p-relation, as many other query answering problems in data management. Thus, considering p-relations is a natural and reasonable restriction for our framework.

The problems studied in this work can be formalized as follows in the framework presented:

Problem:	ENUM(R)
Input:	A word $x \in \Sigma^*$
Output:	Enumerate all $y \in W_R(x)$ without repetitions

Problem:	COUNT(R)
Input:	A word $x \in \Sigma^*$
Output:	The size $|W_R(x)|$

Problem:	GEN(R)
Input:	A word $x \in \Sigma^*$
Output:	Generate uniformly, at random, a word in $W_R(x)$

Given that $|y| = q(|x|)$ for every $(x, y) \in R$, for a polynomial q, we have that $W_R(x)$ is finite and these three problems are well defined. Notice that in the case of ENUM(R), we do not assume a specific order on words, so that the elements of $W_R(x)$ can be enumerated in any order (but without repetitions). Moreover, in the case of COUNT(R), we assume that $|W_R(x)|$ is encoded in binary and, therefore, the size of the output is logarithmic in the size of $W_R(x)$. Finally, in the case of GEN(R), we generate a word $y \in W_R(x)$ with probability $\frac{1}{|W_R(x)|}$ if the set $W_R(x)$ is not empty; otherwise, we return a special symbol $\perp$ to indicate that $W_R(x) = \varnothing$. Hence, in our running example, the problems of enumerating, counting and uniformly generating the answers to a path query correspond to ENUM(EVAL-PQ), COUNT(EVAL-PQ), and GEN(EVAL-PQ), respectively.

In what follows, we present the notions of efficiency that we pursue for the problems studied in this work.

2.1 Notions of efficiency for enumeration

An enumeration algorithm for ENUM(R) is a procedure that receives an input $x \in \Sigma^*$ and, during the computation, it outputs each word in $W_R(x)$, one by one and without repetitions. The time between two consecutive outputs is called the delay of the enumeration. In this paper, we consider two restrictions on the delay: polynomial-delay and constant-delay. *Polynomial-delay enumeration* is the standard notion of polynomial time efficiency in enumeration algorithms [25] and is defined as follows. An enumeration algorithm is of polynomial delay if there exists a polynomial p such that for every input $x \in \Sigma^*$, the time between the beginning of the algorithm and the initial output, between any two consecutive outputs, and between the last output and the end of the algorithm, is bounded by $p(|x|)$.

Constant-delay enumeration is another notion of efficiency for enumeration algorithms that has attracted a lot attention [11, 15, 33]. This notion has stronger guarantees compared to polynomial delay: after the processing of the input, the enumeration is done in a second phase taking constant-time between two consecutive outputs. Several notions of constant-delay enumeration have been studied, most of them in database theory where it is important to divide the analysis between query and data. In this paper, we want a definition of constant-delay that is agnostic of the distinction between query and data (i.e. combined complexity [35]) and, for this reason, we use a more general notion of constant-delay enumeration than the ones in [11, 15, 33].

As it is standard in the literature [33], for constant-delay enumeration we consider enumeration algorithms on Random Access Machines (RAM) with addition and uniform cost measure [4]. Given a relation $R \subseteq \Sigma^* \times \Sigma^*$, an enumeration algorithm $\mathcal{E}$ for R has constant-delay if $\mathcal{E}$ runs in two phases over the input x.

1. The first phase (precomputation), which does not produce output.

2. The second phase (enumeration), which occurs immediately after the precomputation phase, where all words in $W_R(x)$ are enumerated without repetitions and satisfying the following conditions, for a fixed constant c:

   (a) the time it takes to generate the first output y is bounded by $c \cdot |y|$;

(b) the time between two consecutive outputs y and y' is bounded by $c \cdot |y'|$ and does not depend on y; and

(c) the time between the final element y that is returned and the end of the enumeration phase is bounded by $c \cdot |y|$,

We say that $\mathcal{E}$ is a constant-delay algorithm for R with precomputation time f, if $\mathcal{E}$ has constant-delay and the precomputation phase takes time $O(f(|x|))$. Moreover, we say that ENUM(R) can be solved with constant-delay if there exists a constant-delay algorithm for R with precomputation time p for some polynomial p.

Our notion of constant-delay algorithm differs from the definitions in [33] in two aspects. First, as was previously mentioned, we relax the distinction between query and data in the preprocessing phase, allowing our algorithm to take polynomial time in the input (that is, we consider the combined complexity of the problem [35]). Second, our definition of constant-delay is what in [15, 11] is called *linear delay in the size of the output*, namely, writing the next output is linear in its size and does not depend on the size of the input. This is a natural assumption, since each output must at least be written down to return it to the user. Notice that, given an input x and an output y, the notion of polynomial-delay above means polynomial in $|x|$ and, instead, the notion of linear delay from [15, 11] means linear in $|y|$, i.e., constant in the size of $|x|$. Thus, we have decided to call the two-phase enumeration from above "constant-delay", as it does not depend on the size of the input x, and the delay is just what is needed to write the output (which is the minimum requirement for such an enumeration algorithm).

2.2 Notions of efficiency for counting and uniform generation

Given a relation $R \subseteq \Sigma^* \times \Sigma^*$, the problem COUNT$(R)$ can be solved efficiently if there exists a polynomial-time algorithm that, given $x \in \Sigma^*$, computes $|W_R(x)|$. In other words, if we think of COUNT(R) as a function that maps x to the value $|W_R(x)|$, then COUNT(R) can be computed efficiently if COUNT$(R) \in$ FP, the class of functions that can be computed in polynomial time. As such a condition does not hold for many fundamental problems in data management, we also consider the possibility of efficiently approximating the value of the function COUNT(R). More precisely, COUNT(R) is said to admit a fully polynomial-time randomized approximation scheme (FPRAS) [24] if there exists a randomized algorithm $\mathcal{A} : \Sigma^* \times (0,1) \to \mathbb{N}$ and a polynomial $q(u,v)$ such that for every $x \in \Sigma^*$ and $\delta \in (0,1)$, it holds that:

$$\mathbf{Pr}(\,|\mathcal{A}(x,\delta) - |W_R(x)||\; \leq\; \delta \cdot |W_R(x)|\,) \quad \geq \quad \frac{3}{4}$$

and the number of steps needed to compute $\mathcal{A}(x,\delta)$ is at most $q(|x|, 1/\delta)$. Thus, algorithm $\mathcal{A}(x,\delta)$ approximates the value $|W_R(x)|$ with a relative error of δ, and it can be computed in polynomial time in the size of x and the value $1/\delta$.

The problem GEN(R) can be solved efficiently if there exists a polynomial-time randomized algorithm that, given $x \in \Sigma^*$, generates an element of $W_R(x)$ with uniform probability distribution (if $W_R(x) = \varnothing$, then it returns $\perp$). However, as in the case of COUNT(R), the existence of such a generator is not guaranteed for many fundamental problems, so we also consider a relaxed notion of generation that has a probability of failing in returning a solution. More precisely, GEN(R) is said to admit a preprocessing polynomial-time Las Vegas uniform generator (PPLVUG) if there exist a pair of randomized algorithms $\mathcal{P} : \Sigma^* \times (0,1) \to (\Sigma^* \cup \{\perp\})$, $\mathcal{G} : \Sigma^* \to (\Sigma^* \cup \{\mathbf{fail}\})$, a set $\mathcal{V} \subseteq \Sigma^*$, and a pair of polynomials $q(u,v)$, $r(u)$ such that for every $x \in \Sigma^*$ and $\delta \in (0,1)$:

1. The preprocessing algorithm $\mathcal{P}$ receives as inputs x and δ and performs at most $q(|x|, \log(1/\delta))$ steps. If $W_R(x) \neq \varnothing$, then $\mathcal{P}(x,\delta)$ returns a string $\mathbf{d}$ such that $\mathbf{d} \in \mathcal{V}$ with probability $1-\delta$. If $W_R(x) = \varnothing$, then $\mathcal{P}(x,\delta)$ returns $\perp$.

2. The generator algorithm $\mathcal{G}$ receives as input $\mathbf{d}$ and performs at most $r(|\mathbf{d}|)$ steps. Moreover, if $\mathbf{d} \in \mathcal{V}$, then:

   (a) $\mathcal{G}(\mathbf{d})$ returns **fail** with a probability of at most $\frac{1}{2}$, and

   (b) conditioned on not returning **fail**, $\mathcal{G}(\mathbf{d})$ returns a truly uniform sample $y \in W_R(x)$, i.e. with a probability $1/|W_R(x)|$ for each $y \in W_R(x)$.

   Otherwise, if $\mathbf{d} \notin \mathcal{V}$, then $\mathcal{G}(\mathbf{d})$ outputs a string without any guarantee.

The set $\mathcal{V}$ of strings is called the set of *valid* strings. In line with the notion of constant-delay enumeration algorithm, we allow the previous concept of uniform generator to have a preprocessing phase. If there is no witness for the input x (that is, $W_R(x) = \varnothing$), then the preprocessing algorithm $\mathcal{P}$ returns the symbol $\perp$. Otherwise, the invocation $\mathcal{P}(x,\delta)$ returns a string $\mathbf{d}$ in Σ^*, namely, a data structure or "advice" for the generation procedure $\mathcal{G}$. The output of the invocation $\mathcal{P}(x,\delta)$ is used by the generator algorithm $\mathcal{G}$ to produce a witness of x with uniform distribution (that is, with probability $1/|W_R(x)|$). If the output of $\mathcal{P}(x,\delta)$ is not valid (which occurs with probability δ), then we have no guarantees on the output of the generator algorithm $\mathcal{G}$. Otherwise, we know that $\mathcal{G}(\mathbf{d})$ returns an element of $W_R(x)$ with uniform distribution, or it returns **fail**. Furthermore, we can repeat $\mathcal{G}(\mathbf{d})$ as many times as needed, generating each time a truly uniform sample y from $W_R(x)$ whenever $y \neq$ **fail**. It is important to notice that the definition of PPLVUG does not guarantee that it can be distinguished in polynomial time whether $\mathbf{d}$ is valid (that is, whether $\mathbf{d} \in \mathcal{V}$), so $\mathcal{G}$ has to use $\mathbf{d}$ only knowing that with probability $1-\delta$ is a valid string, in which case $\mathbf{d}$ will be useful for generating an element of $W_R(x)$ with uniform distribution.

Notice that by condition (2a), we know that the probability of failing is smaller than $\frac{1}{2}$, so that by calling $\mathcal{G}(\mathbf{d})$ several times we can make this probability arbitrarily small. For example, the probability that $\mathcal{G}(\mathbf{d})$ returns **fail** in 1000 consecutive independent invocations is at most $(\frac{1}{2})^{1000}$. Furthermore, we have that $\mathcal{P}(x,\delta)$ can be computed in time $q(|x|, \log(1/\delta))$, so we can consider an exponentially small value of δ such as

$$\frac{1}{2^{|x|+1000}}$$

and still obtain that $\mathcal{P}(x,\delta)$ can be computed in time polynomial in $|x|$. Notice that with such a value of δ, the probability of producing a valid string $\mathbf{d}$ is at least

$$1 - \frac{1}{2^{1000}}$$

which is an extremely high probability. Finally, it is important to notice that the size of $\mathbf{d}$ is at most $q(|x|, \log(1/\delta))$, so that $\mathcal{G}(\mathbf{d})$ can be computed in time polynomial in $|x|$ and $\log(1/\delta)$. Therefore, $\mathcal{G}(\mathbf{d})$ can be computed in time polynomial in $|x|$ even if we consider an exponentially small value for δ such as $1/2^{|x|+1000}$.

Notice that the notion of preprocessing polynomial-time Las Vegas uniform generator imposes stronger requirements than the notion of fully polynomial-time almost uniform generator introduced in [24]. In particular, the latter not only has a probability of failing, but also considers the possibility of generating a solution with a probability distribution that is *almost* uniform, that is, an algorithm that generates a string $y \in W_R(x)$ with a probability in an interval $[1/|W_R(x)| - \varepsilon, 1/|W_R(x)| + \varepsilon]$ for a given error $\varepsilon \in (0,1)$.

3. OUR MAIN CONTRIBUTIONS

The goal of this section is to provide simple yet general definitions of classes of relations with good properties in terms of enumeration, counting, and uniform generation. More precisely, we are first aiming at providing a class $\mathcal{C}$ of relations that has a simple definition in terms of Turing Machines and such that for every relation $R \in \mathcal{C}$, it holds that ENUM(R) can be solved with constant delay, and both COUNT(R) and GEN(R) can be solved in polynomial time. Moreover, as it is well known that such good conditions cannot always be achieved, we are then aiming at extending the definition of $\mathcal{C}$ to obtain a simple class, also defined in terms of Turing Machines and with good approximation properties. It is important to mention that we are not looking for an exact characterization in terms of Turing Machines of the class of relations that admit constant delay enumeration algorithms, as this may result in an overly complicated model. Instead, we are looking for simple yet general classes of relations with good properties in terms of enumeration, counting, and uniform generation, and which can serve as a starting point for the systematic study of these three fundamental properties together.

3.1 The general class RelationNL

A key notion that is used in our definitions of classes of relations is that of a transducer. Given a finite alphabet Σ, an NL-transducer M is a nondeterministic Turing Machine with input and output alphabet Σ, a read-only input tape, a write-only output tape where the head is always moved to the right once a symbol is written in it (so that the output cannot be read by M), and a work-tape of which, on input x, only the first $f(|x|)$ cells can be used, where $f(n) \in O(\log(n))$. A string $y \in \Sigma^*$ is said to be an output of M on input x, if there exists a run of M on input x that halts in an accepting state with y as the string in the output tape. The set of all outputs of M on input x is denoted by $M(x)$ (notice that $M(x)$ can be empty). Finally, the relation accepted by M, denoted by $\mathcal{R}(M)$, is defined as $\{(x,y) \in \Sigma^* \times \Sigma^* \mid y \in M(x)\}$.

Definition 3.1. *A relation R is in* RelationNL *iff there exists an* NL*-transducer M such that $\mathcal{R}(M) = R$.*

The class RelationNL should be general enough to contain some natural and well-studied problems. A first such a problem is the satisfiability of a propositional formula in DNF. This problem can be naturally represented as follows:

$$\begin{aligned}\text{SAT-DNF} = \{(\varphi,\sigma) \mid\ & \varphi \text{ is a proposional formula} \\ & \text{in DNF, } \sigma \text{ is a truth assignment and } \sigma(\varphi) = 1\}.\end{aligned}$$

Thus, we have that ENUM(SAT-DNF) corresponds to the problem of enumerating the truth assignments satisfying a propositional formula φ in DNF, while COUNT(SAT-DNF) and GEN(SAT-DNF) correspond to the problems of counting and uniformly generating such truth assignments, respectively. It is not difficult to see that SAT-DNF is in RelationNL. In fact, assume that we are given a propositional formula φ of the form $D_1 \vee \cdots \vee D_m$, where each D_i is a conjunction of literals, that is, a conjunction of propositional variables and negation of propositional variables. Moreover, assume that each propositional variable in φ is of the form x_k, where k is a binary number, and that x_1, ..., x_n are the variables occurring in φ. Notice that with such a representation, we have that φ is a string over the alphabet $\{x, \_, 0, 1, \wedge, \vee, \neg\}$. We define as follows an NL-transducer M such that $M(\varphi)$ is the set of truth assignments satisfying φ. On input φ, the NL-transducer M non-deterministically chooses a disjunct D_i, which is represented by two indexes indicating the starting and ending symbols of D_i in the string φ. Then it checks whether D_i is satisfiable, that is, whether D_i does not contain complementary literals. Notice that this can be done in logarithmic space by checking for every $j \in \{1, \ldots, n\}$, whether x_j and $\neg x\_j$ are both literals in D_i. If D_i is not satisfiable, then M halts in a non-accepting state. Otherwise, M returns a satisfying truth assignment of D_i as follows. A truth assignment for φ is represented by a string of length n over the alphabet $\{0,1\}$, where the j-th symbol of this string is the truth value assigned to variable x_j. Then for every $j \in \{1, \ldots, n\}$, if x_j is a conjunct in D_i, then M write the symbol 1 in the output tape, and if $\neg x\_j$ is a conjunct in D_i, then M write the symbol 0 in the output tape. Finally if neither x_j nor $\neg x\_j$ is a conjunct in D_i, then M non-deterministically chooses a symbol $b \in \{0,1\}$, and it writes b in the output tape.

By using NL-transducers, one can easily show that some query answering problems in data management are also in RelationNL, in the same way as for the case of SAT-DNF. For instance, the relation EVAL-PQ defined in (†), which is used to encode our running example, can be shown to be in RelationNL. To see this, assume a reasonable encoding for an input (G, Q, n) (this time, we omit the string representation of the input and output for simplicity). In particular, assume that $Q = (a, r, b)$ is a path query with a, b vertices in G and r a regular expression. Then our NL-transducer constructs on-the-fly the product of G with an NFA A accepting the regular language defined by r, uses this product to traverse G through a path π such that π conforms to r and $|\pi| = n$, and outputs π. More precisely, our NL-transducer keeps a counter c and two indices, called g and q, pointing to a vertex in G and a state in A, respectively. The transducer starts with $c = 0$, $g = a$ and $q = q_0$, assuming that q_0 is the initial state of A. Then, at each step the machine non-deterministically chooses an edge (g, ℓ, g') on G and a transition (q, ℓ, q') on A for some edge-label ℓ, writes (g, ℓ, g') in the output tape, and updates c, g, and q to $c+1$, g', and q', respectively. If this combination edge-transition does not exist or c becomes greater than n, then the machine stops and rejects. Instead, if it holds that g is equal to b, q is a final state of A, and $c = n$, then our NL-transducer stops and accepts. In other words, we have shown that EVAL-PQ $\in$ RELATIONNL.

The problem COUNT(SAT-DNF) is a paramount example of a #P-complete problem. Moreover, it is known that COUNT(EVAL-PQ) is #P-complete as well [9]. Hence, we cannot expect COUNT(R) to be solvable in polynomial time for every $R \in$ RELATIONNL. However, the problem COUNT(SAT-DNF) admits an FPRAS [27], so we can still hope for COUNT(R) to admit an FPRAS for every $R \in$ RELATIONNL. In this work, we give a positive answer to the question of the existence of such an approximation algorithm for every relation in RELATIONNL.

Theorem 3.2. *If $R \in$ RELATIONNL, then ENUM(R) can be solved with polynomial delay, COUNT(R) admits an FPRAS, and GEN(R) admits a PPLVUG.*

In particular, given that EVAL-PQ is in RELATIONNL, the three problems for graph databases mentioned in Sections 1 and 2 have good algorithmic properties: ENUM(EVAL-PQ) can be solved with polynomial delay, COUNT(EVAL-PQ) admits an FPRAS, and GEN(EVAL-PQ) admits a PPLVUG. Notice that deriving a polynomial-delay enumeration algorithm for EVAL-PQ is straightforward, but the existence of an FPRAS for COUNT(EVAL-PQ), as well as of a PPLVUG for GEN(EVAL-PQ), was not known before. This is one of the main advantages of our approach: by proving membership in RELATIONNL, we can easily identify query answering problems that have good algorithmic properties in terms of enumeration, counting, and uniform generation.

3.1.1 A fundamental consequence of our result

It turns out that proving our main result (Theorem 3.2) involves providing an FPRAS for a natural problem associated to path queries and graph databases. More specifically, #NFA is the problem of counting the number of words of length k accepted by a non-deterministic finite automaton (NFA), where k is given in unary (that is, k is given as a string 0^k). It is known that #NFA is #P-complete [5], but it is open whether it admits an FPRAS; in fact, the best randomized approximation scheme known for #NFA runs in time $n^{O(\log(n))}$ [26]. In our notation, this problem is represented by the following relation:

$$\begin{aligned}\text{MEM-NFA} \;=\; & \{((A, 0^k), w) \mid A \text{ is an NFA and} \\ & w \text{ is a word of length } k \text{ accepted by } A\},\end{aligned}$$

that is, #NFA is the same problem as COUNT(MEM-NFA). To prove Theorem 3.2, we have to provide an FPRAS for COUNT(MEM-NFA), thus giving a positive answer to the open question of whether #NFA admits an FPRAS.

It is important to mention a fundamental consequence of this result in computational complexity. The class of function SPANL was introduced in [5] to provide a characterization of some functions that are hard to compute. More specifically, given a finite alphabet Σ, a function $f : \Sigma^* \to \mathbb{N}$ is in SPANL if there exists an NL-transducer M with input alphabet Σ such that $f(x) = |M(x)|$ for every $x \in \Sigma^*$. The complexity class SPANL is contained in #P, and it is a hard class in the sense that if SPANL $\subseteq$ FP, then P = NP [5], where FP is the class of functions that can be computed in polynomial time. In fact, SPANL has been instrumental in proving that some functions are difficult to compute [5, 22, 9, 28].

It is easy to see that #NFA belongs to SPANL. In fact, it was shown in [5] that #NFA is SPANL-complete under the notion of parsimonious reduction. Therefore, given that a parsimonious reduction preserves the existence of an FPRAS, we obtain the following corollary from Theorem 3.2 and our characterization of #NFA as COUNT(MEM-NFA):

Corollary 3.3. *Every function in SPANL admits an FPRAS.*

Although some classes $\mathcal{C}$ containing #P-complete functions and for which every $f \in \mathcal{C}$ admits an FPRAS have been identified before [32, 10], to the best of our knowledge this is the first such a class with a simple and robust definition based on Turing Machines.

3.2 The more restricted class RelationUL

A natural question at this point is whether a simple syntactic restriction on the definition of RELATIONNL gives rise to a class of relations with better properties in terms of enumeration, counting, and uniform generation. Fortunately, the answer to this question comes by imposing a natural and well-studied restriction on Turing Machines, which allows us to define a class that contains many natural problems. More precisely, we consider the notion of UL-transducer, where the letter "U" stands for "unambiguous". Formally, M is a UL-transducer if M is an NL-transducer such that for every input x and $y \in M(x)$, there exists exactly one run of M on input x that halts in an accepting state with y as the string in the output tape. Notice that this notion of transducer is based on well-known classes of decision problems (e.g. UP [34] and UL [31]) adapted to our case, namely, adapted to problems defined as relations.

Definition 3.4. *A relation R is in* RELATIONUL *iff there exists a* UL*-transducer M such that $\mathcal{R}(M) = R$.*

For the class RELATIONUL, we obtain the following result.

Theorem 3.5. *If $R \in$ RELATIONUL, then ENUM(R) can be solved with constant delay, there exists a polynomial-time algorithm for COUNT(R), and there exists a polynomial-time randomized algorithm for GEN(R).*

Hence, given a relation R in RELATIONUL and an input x, the solutions for x can be enumerated, counted and uniformly generated efficiently. Classes of problems definable by machine models and that can be enumerated with constant delay have been proposed before. In [6], it is shown that if a problem is definable by a d-DNNF circuit, then the solutions of an instance can be listed with linear preprocessing and constant delay enumeration. Still, to the best of our knowledge, this is the first such a class with a simple and robust definition based on Turing Machines.

4. OTHER APPLICATIONS OF THE MAIN RESULTS

By using our machinery, we have already proved that query evaluation in graph databases has good properties in terms of enumeration, approximate counting, and uniform generation. In this section, we show further applications of our main results in information extraction and binary decision diagrams.

4.1 Information extraction

In [16], the framework of document spanners was proposed as a formalization of ruled-based information extraction. In this framework, the main data objects are documents and spans. Formally, given a finite alphabet Σ, a document is a string $d = a_1 \dots a_n$ and a span is a pair $s = [i, j\rangle$ with $1 \le i \le j \le n+1$. A span represents a continuous region of the document d, whose content is the substring of d from positions i to $j-1$. Given a finite set of variables $\mathbf{X}$, a mapping μ is a function from $\mathbf{X}$ to the spans of d.

Variable set automata (VA) are one of the main formalisms to specify sets of mappings over a document. Here, we use the notion of extended VA (eVA) from [17] to state our main results. We only recall the main definitions, and we refer the reader to [17, 16] for more intuition and further details. An eVA is a tuple $\mathcal{A} = (Q, q_0, F, \delta)$ such that Q is a finite set of states, q_0 is the initial state, and F is the final set of states. Further, δ is the transition relation consisting of letter transitions (q, a, q'), or variable-set transitions (q, S, q'), where $S \subseteq \{x\vdash, \dashv x \mid x \in \mathbf{X}\}$ and $S \neq \varnothing$. The symbols $x\vdash$ and $\dashv x$ are called markers, and they are used to denote that variable x is open or close by $\mathcal{A}$, respectively. A run ρ over a document $d = a_1 \cdots a_n$ is a sequence of the form: $q_0 \xrightarrow{X_1} p_0 \xrightarrow{a_1} q_1 \xrightarrow{X_2} p_1 \xrightarrow{a_2} \dots \xrightarrow{a_n} q_n \xrightarrow{X_{n+1}} p_n$ where each X_i is a (possible empty) set of markers, $(p_i, a_{i+1}, q_{i+1}) \in \delta$, and $(q_i, X_{i+1}, p_i) \in \delta$ whenever $X_{i+1} \neq \varnothing$, and $q_i = p_i$ otherwise (that is, when $X_{i+1} = \varnothing$). We say that a run ρ is valid if for every $x \in \mathbf{X}$ there exists exactly one pair $[i, j\rangle$ such that $x\vdash \in X_i$ and $\dashv x \in X_j$. A valid run ρ naturally defines a mapping μ^ρ that maps x to the only span $[i, j\rangle$ such that $x\vdash \in X_i$ and $\dashv x \in X_j$. We say that ρ is accepting if $p_n \in F$. Finally, the semantics $[\![\mathcal{A}]\!](d)$ of $\mathcal{A}$ over d is defined as the set of all mappings μ^ρ where ρ is a valid and accepting run of $\mathcal{A}$ over d.

In [19, 30], it was shown that the decision problem related to query evaluation, namely, given an eVA $\mathcal{A}$ and a document d deciding whether $[\![\mathcal{A}]\!](d) \neq \varnothing$, is NP-hard. For this reason, in [17] a subclass of eVA is considered in order to recover polynomial-time evaluation. An eVA $\mathcal{A}$ is called functional if every accepting run is valid. Intuitively, a functional eVA does not need to check validity of the run given that it is already known that every run that reaches a final state will be valid. For the query evaluation problem of functional eVA (i.e. to compute $[\![\mathcal{A}]\!](d)$), one can naturally associate the following relation:

$$\begin{aligned}\text{EVAL-eVA} = \{((\mathcal{A}, d), \mu) \mid {} & \mathcal{A} \text{ is a functional eVA}, \\ & d \text{ is a document, and } \mu \in [\![\mathcal{A}]\!](d)\}\end{aligned}$$

It is not difficult to show that EVAL-eVA $\in$ RELATIONNL. Hence, by Theorem 3.2 we get the following results.

Corollary 4.1. *ENUM(EVAL-eVA) can be enumerated with polynomial delay, COUNT(EVAL-eVA) admits an FPRAS, and GEN(EVAL-eVA) admits a PPLVUG.*

In [17], it was shown that every functional RGX or functional VA (not necessarily extended) can be converted in polynomial time into a functional eVA. Therefore, Corollary 4.1 also holds for these more general classes.

Regarding efficient enumeration and exact counting, an algorithm for constant-delay enumeration was given in [17] for the class of deterministic functional eVA. Here, we can extend these results for a more general class, that we called unambiguous functional eVA. Formally, we say that an eVA is unambiguous if for every two valid and accepting runs ρ_1 and ρ_2, it holds that $\mu^{\rho_1} \neq \mu^{\rho_2}$. In other words, each output of an unambiguous eVA is witness by exactly one run. As in the case of EVAL-eVA, we can define the relation EVAL-UeVA, by restricting the input to unambiguous functional eVA. By using UL-transducers and Theorem 3.5, we can then extend the results in [17] for the unambiguous case.

Corollary 4.2. *ENUM(EVAL-UeVA) can be solved with constant delay, there exists a polynomial-time algorithm for COUNT(EVAL-UeVA), and there exists a polynomial-time randomized algorithm for GEN(EVAL-UeVA).*

4.2 Binary decision diagrams

Binary decision diagrams are an abstract representation of boolean functions which are widely used in computer science and have found many applications in areas like formal verification [13]. A binary decision diagram (BDD) is a directed acyclic graph $D = (V, E)$ where each node v is labeled with a variable $\text{var}(v)$ and has at most two edges going to children $\text{lo}(v)$ and $\text{hi}(v)$. Intuitively, $\text{lo}(v)$ and $\text{hi}(v)$ represent the next nodes when $\text{var}(v)$ takes values 0 and 1, respectively. D contains only two terminal, or sink nodes, labeled by 0 or 1, and one initial node called v_0. We assume that every path from v_0 to a terminal node does not repeat variables. Then given an assignment σ from the variables in D to $\{0, 1\}$, we have that σ naturally defines a path from v_0 to a terminal node 0 or 1. In this way, D defines a boolean function that gives a value in $\{0, 1\}$ to each assignment σ; in particular, $D(\sigma) \in \{0, 1\}$ corresponds to the sink node reached by starting from v_0 and following the values in σ. For Ordered BDDs (OBDDs), we also have a linear order $<$ over the variables in D such that, for every $v_1, v_2 \in V$ with v_2 a child of v_1, it holds that $\text{var}(v_1) < \text{var}(v_2)$. Note that not all variables need to appear in a path from the initial node v_0 to a terminal node 0 or 1. Nevertheless, the promise in an OBDD is that variables will appear following the order $<$.

An OBDD D defines the set of assignments σ such that $D(\sigma) = 1$. Then D can be considered as a succinct representation of the set $\{\sigma \mid D(\sigma) = 1\}$, and one would like to enumerate, count and uniformly generate assignments given D. This motivates the use of the relation:

$$\text{EVAL-OBDD} = \{(D, \sigma) \mid D(\sigma) = 1\}.$$

Given (D, σ) in EVAL-OBDD, there is exactly one path in D that witnesses $D(\sigma) = 1$. Therefore, one can easily show that EVAL-OBDD is in RELATIONUL.

Corollary 4.3. *ENUM(EVAL-OBDD) can be enumerated with constant delay, there exists a polynomial-time algorithm for COUNT(EVAL-OBDD), and there exists a polynomial-time randomized algorithm for GEN(EVAL-OBDD).*

The above results are well known. However, they show how easy and direct is to use UL-transducers to obtain some of the good algorithmic properties of OBDDs.

Some non-deterministic variants of BDDs have been studied [7]. In particular, an nOBDD extends an OBDD with vertices u without variables (i.e. $\text{var}(u) = \perp$) and without labels on its children. Thus, an nOBDD is non-deterministic in the sense that given an assignment σ, there can be several paths that bring σ from the initial node v_0 to a terminal node with labeled 0 or 1. Without lost of generality, nOBDDs are assumed to be consistent in the sense that, for each σ, all paths of σ in D can reach 0 or 1, but not both.

As in the case of OBDDs, we can define EVAL-nOBDD that pairs an nOBDD D with an assignment σ that evaluate D to 1 (i.e. $D(\sigma) = 1$). Contrary to OBDDs, an nOBDD looses the single witness property, and now an assignment σ can have several paths from the initial node to the 1 terminal node. Thus, it is not clear whether EVAL-nOBDD is in RelationUL. Still one can easily show that EVAL-nOBDD is in RelationNL, from which the following results follow.

Corollary 4.4. *ENUM(EVAL-nOBDD) can be solved with polynomial delay, COUNT(EVAL-nOBDD) admits an FPRAS, and GEN(EVAL-nOBDD) admits a PPLVUG.*

It should be noticed that the existence of an FPRAS and a PPLVUG for EVAL-nOBDD was not known before, and one can easily show this by using NL-transducers and then applying Theorem 3.2.

5. CONCLUDING REMARKS

We consider this work as a first step towards the definition of classes of problems in data management with good properties in terms of enumeration, counting, and uniform generation of solutions. Given the relevance of these problems for query answering, identifying good complexity classes, like RelationNL and RelationUL, should be the cornerstone to better understand the complexity of query evaluation. In this sense, there is plenty of room for extensions and improvements. In particular, one could be more ambitious and ask for more conditions to these relations, like having good properties in terms of ranked enumeration [33] (i.e. enumeration of the solutions following some specific order) or random generation with respect to a user-defined distribution. Moreover, we believe that other classes with good algorithmic properties can be identified, which could serve to unify enumeration, counting, and uniform generation in data management.

6. REFERENCES

[1] S. Abiteboul and G. Dowek. *The Age of Algorithms*. Cambridge University Press, 2020.

[2] S. Abiteboul, G. Miklau, J. Stoyanovich, and G. Weikum. Data, responsibly. *Dagstuhl Reports*, 6(7):42–71, 2016.

[3] S. Acharya, P. B. Gibbons, V. Poosala, and S. Ramaswamy. Join synopses for approximate query answering. In *SIGMOD*, pages 275–286, 1999.

[4] A. V. Aho and J. E. Hopcroft. *The design and analysis of computer algorithms*. Pearson Education India, 1974.

[5] C. Álvarez and B. Jenner. A very hard log-space counting class. *Theoretical Computer Science*, 107(1):3–30, 1993.

[6] A. Amarilli, P. Bourhis, L. Jachiet, and S. Mengel. A circuit-based approach to efficient enumeration. In *Proceedings of ICALP*, pages 111:1–111:15, 2017.

[7] A. Amarilli, F. Capelli, M. Monet, and P. Senellart. Connecting knowledge compilation classes and width parameters. *CoRR*, abs/1811.02944, 2018.

[8] R. Angles, M. Arenas, P. Barceló, A. Hogan, J. Reutter, and D. Vrgoč. Foundations of modern query languages for graph databases. *ACM Computing Surveys*, 50(5):68, 2017.

[9] M. Arenas, S. Conca, and J. Pérez. Counting beyond a yottabyte, or how SPARQL 1.1 property paths will prevent adoption of the standard. In *WWW*, pages 629–638, 2012.

[10] M. Arenas, M. Muñoz, and C. Riveros. Descriptive complexity for counting complexity classes. In *LICS*, pages 1–12, 2017.

[11] G. Bagan. MSO queries on tree decomposable structures are computable with linear delay. In *Proceedings of CSL*, pages 167–181, 2006.

[12] G. Bagan, A. Durand, and E. Grandjean. On acyclic conjunctive queries and constant delay enumeration. In *Proceedings of CSL*, pages 208–222, 2007.

[13] R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys (CSUR)*, 24(3):293–318, 1992.

[14] S. Chaudhuri, R. Motwani, and V. R. Narasayya. On random sampling over joins. In *SIGMOD*, pages 263–274, 1999.

[15] B. Courcelle. Linear delay enumeration and monadic second-order logic. *Discrete Applied Mathematics*, 157(12):2675–2700, 2009.

[16] R. Fagin, B. Kimelfeld, F. Reiss, and S. Vansummeren. Document spanners: A formal approach to information extraction. *Journal of the ACM*, 62(2):12, 2015.

[17] F. Florenzano, C. Riveros, M. Ugarte, S. Vansummeren, and D. Vrgoc. Efficient enumeration algorithms for regular document spanners. *ACM Trans. Database Syst.*, 45(1):3:1–3:42, 2020.

[18] N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, P. Selmer, and A. Taylor. Cypher: An evolving query language for property graphs. In *Proceedings of SIGMOD*, pages 1433–1445, 2018.

[19] D. D. Freydenberger. A logic for document spanners. In *Proceedings of ICDT*, pages 13:1–13:18, 2017.

[20] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database systems - the complete book*. Pearson Education, 2009.

[21] V. Gore, M. Jerrum, S. Kannan, Z. Sweedyk, and S. R. Mahaney. A quasi-polynomial-time algorithm for sampling words from a context-free language. *Inf. Comput.*, 134(1):59–74, 1997.

[22] L. A. Hemaspaandra and H. Vollmer. The satanic notations: counting classes beyond #P and other definitional adventures. *SIGACT News*, 26(1):2–13, 1995.

[23] Y. E. Ioannidis. The history of histograms (abridged). In *VLDB*, pages 19–30. Morgan Kaufmann, 2003.

[24] M. R. Jerrum, L. G. Valiant, and V. V. Vazirani. Random generation of combinatorial structures from a uniform distribution. *Theor. Comput. Sci.*, 43:169–188, 1986.

[25] D. S. Johnson, M. Yannakakis, and C. H. Papadimitriou. On generating all maximal independent sets. *Information Processing Letters*, 27(3):119–123, 1988.

[26] S. Kannan, Z. Sweedyk, and S. R. Mahaney. Counting and random generation of strings in regular languages. In *Proceedings of SODA*, pages 551–557, 1995.

[27] R. M. Karp and M. Luby. Monte-carlo algorithms for enumeration and reliability problems. In *Proceedings of FOCS*, pages 56–64, 1983.

[28] K. Losemann and W. Martens. The complexity of regular expressions and property paths in SPARQL. *ACM Trans. Database Syst.*, 38(4):24:1–24:39, 2013.

[29] W. Martens and T. Trautner. Dichotomies for evaluating simple regular path queries. *ACM Trans. Database Syst.*, 44(4):16:1–16:46, 2019.

[30] F. Maturana, C. Riveros, and D. Vrgoc. Document spanners for extracting incomplete information: Expressiveness and complexity. In *Proceedings of PODS*, pages 125–136. ACM, 2018.

[31] K. Reinhardt and E. Allender. Making nondeterminism unambiguous. *SIAM J. Comput.*, 29(4):1118–1131, 2000.

[32] S. Saluja, K. Subrahmanyam, and M. N. Thakur. Descriptive complexity of #P functions. *Journal of Computer and System Sciences*, 50(3):493–505, 1995.

[33] L. Segoufin. Enumerating with constant delay the answers to a query. In *Proceedings of ICDT*, pages 10–20, 2013.

[34] L. G. Valiant. Relative complexity of checking and evaluating. *Inf. Process. Lett.*, 5(1):20–23, 1976.

[35] M. Y. Vardi. The complexity of relational query languages (extended abstract). In *STOC*, pages 137–146, 1982.

[36] Z. Zhao, R. Christensen, F. Li, X. Hu, and K. Yi. Random sampling over joins revisited. In *SIGMOD*, pages 1525–1539. ACM, 2018.

Technical Perspective: Query Optimization for Faster Deep CNN Explanations

Sebastian Schelter
University of Amsterdam
s.schelter@uva.nl

Machine learning (ML) is increasingly used to automate decision making in various domains. In recent years, ML has not only been applied to tasks that use structured input data, but also, tasks that operate on data with less strictly defined structure such as speech, images and videos. Prominent examples are speech recognition for personal assistants or face recognition for boarding airplanes.

Responsible data management. There exists a variety of challenges with respect to the fairness, accountability and transparency of the resulting automating decision-making systems, whose data specific aspects are addressed by research under the umbrella of "responsible data management" [2]. A particular challenge in this area is the explainability of the predictions of an ML model. Common approaches derive local explanations for a model's predictions by perturbing the features of a single example and calculating how much these perturbations affect the prediction outcome as a measure of the explanatory power of the perturbed feature [7].

Data management for machine learning. ML poses additional challenges apart from responsibility. ML models are part of larger end-to-end ML pipelines, which include the integration, validation, and cleaning of data, as well as the training, deployment and analysis of models. The definition, maintenance and efficient execution of such pipelines pose various data management challenges [4, 6, 8].

Unfortunately, it is often very difficult to apply established data management techniques, such as query optimization or provenance tracking, as they rely on an abstract algebraic specification of the computation, which is typically lacking for end-to-end ML pipelines. These pipelines comprise of the previously mentioned heterogeneous stages, for which different systems are often "glued" together in the real world. This results in tedious work, complex environments, and a loss of potential for optimisation. It is an ongoing research challenge to find well working abstractions for ML computations that incorporate data and operations from both relational algebra and linear algebra [5].

Convolutional neural networks. Deep neural networks are the current state-of-the-art in machine learning, and heavily influence adjacent domains such as computer vision and natural language processing. Convolutional neural networks (CNNs), designed to learn features from image data, started the triumph of deep neural networks with their outstanding performance in image recognition tasks [3], and earned their inventor Yann LeCun a Turing award. Efficient training and inference for deep neural networks is gaining a lot of attention recently, e.g., in the form of specialized optimizing compilers [1].

The highlighted paper. The work by Nakandala et al. concentrates on the problem of efficiently computing occlusion-based explanations of the predictions of a CNN. This explanation method repeatedly occludes small regions of an input image, and measures the corresponding changes in the predictions. This approach requires a very large number of inference requests to the CNN, and the paper presents efficient methods to drastically reduce the runtime of the corresponding computation.

The beauty of this research lies in the fact that it elegantly connects all three previously discussed areas: responsible data management (explaining the predictions of an ML model), deep learning (with its focus on convolutional neural networks) and data management (query optimisation for ML inference).

The paper is based on the observation that deep neural networks open up many opportunities for applying established optimisations from relational query processing, as they also build on a strict algebraic foundation: their underlying computations are modeled as directed acyclic graphs of linear algebra operators, which exchange data in the form of tensors. The authors take established techniques from the data management space (incremental view maintenance and multi-query optimisation), and reinvent them for the ML related context of efficiently executing a large number of related inference requests. For that, the paper treats inference requests as "queries", the CNN as a "query plan" with operators like max-pooling, and tensors (which represent the input images and operator outputs) as "relations".

This work is an important step forward towards bridging the gap between classical relational data management and modern machine learning workloads. I hope that we will see generalizations of the applied techniques for a wide variety of related ML tasks in the future.

REFERENCES

[1] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. *OSDI*, 578–594.
[2] HV Jagadish, Francesco Bonchi, Tina Eliassi-Rad, Lise Getoor, Krishna Gummadi, and Julia Stoyanovich. 2019. The Responsibility Challenge for Data. *SIGMOD*, 412–414.
[3] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet Classification with Deep Convolutional Neural Networks. *NeurIPS*, 1097–1105.
[4] Arun Kumar, Robert McCann, Jeffrey Naughton, and Jignesh M Patel. 2016. Model Selection Management Systems: The Next Frontier of Advanced Analytics. *SIGMOD Record* 44, 4 (2016), 17–22.
[5] Andreas Kunft, Alexander Alexandrov, Asterios Katsifodimos, and Volker Markl. 2016. Bridging the gap: towards optimization across linear and relational algebra. *Workshop on Algorithms and Systems for MapReduce and Beyond at SIGMOD*, 1–4.
[6] Neoklis Polyzotis, Sudip Roy, Steven Euijong Whang, and Martin Zinkevich. 2018. Data Lifecycle Challenges in Production Machine Learning: A Survey. *SIGMOD Record* 47, 2 (2018), 17–28.
[7] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. 2016. "Why should I trust you?" Explaining the Predictions of Any Classifier. *KDD*, 1135–1144.
[8] David Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-Francois Crespo, and Dan Dennison. 2015. Hidden Technical Debt in Machine Learning Systems. *NeurIPS*, 2503–2511.

Query Optimization for Faster Deep CNN Explanations

Supun Nakandala, Arun Kumar, and Yannis Papakonstantinou
University of California, San Diego
{snakanda,arunkk,yannis}@eng.ucsd.edu

ABSTRACT

Deep Convolutional Neural Networks (CNNs) now match human accuracy in many image prediction tasks, resulting in a growing adoption in e-commerce, radiology, and other domains. Naturally, "explaining" CNN predictions is a key concern for many users. Since the internal workings of CNNs are unintuitive for most users, *occlusion-based explanations* (OBE) are popular for understanding which parts of an image matter most for a prediction. One occludes a region of the image using a patch and moves it around to produce a *heatmap* of changes to the prediction probability. This approach is computationally expensive due to the large number of re-inference requests produced, which wastes time and raises resource costs. We tackle this issue by casting the OBE task as a new instance of the classical incremental view maintenance problem. We create a novel and comprehensive algebraic framework for incremental CNN inference combining materialized views with multi-query optimization to reduce computational costs. We then present two novel approximate inference optimizations that exploit the semantics of CNNs and the OBE task to further reduce runtimes. We prototype our ideas in a tool we call KRYPTON. Experiments with real data and CNNs show that KRYPTON reduces runtimes by up to 5x (resp. 35x) to produce exact (resp. high-quality approximate) results without raising resource requirements.

1. INTRODUCTION

Deep Convolutional Neural Networks (CNNs) are now the state-of-the-art machine learning (ML) method for many image prediction tasks [25]. Thus, there is growing adoption of deep CNNs in many applications across healthcare, domain sciences, enterprises, and Web companies. Remarkably, even the US Food and Drug Administration recently approved the use of deep CNNs to assist radiologists in processing X-rays and other scans, cross-checking their decisions, and even mitigating the shortage of radiologists [1].

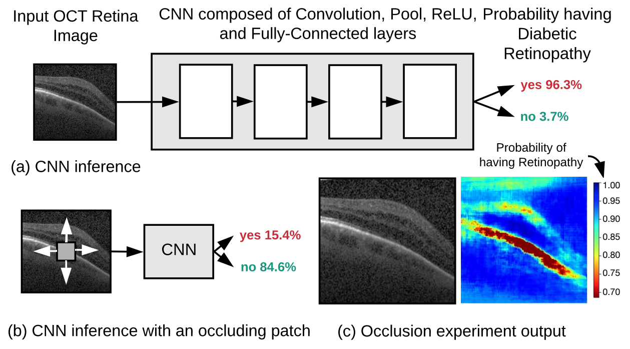

Input OCT Retina Image
CNN composed of Convolution, Pool, ReLU, and Fully-Connected layers
Probability having Diabetic Retinopathy
yes 96.3%
no 3.7%
(a) CNN inference
Probability of having Retinopathy
1.00
0.95
0.90
0.85
0.80
0.75
0.70
CNN
yes 15.4%
no 84.6%
(b) CNN inference with an occluding patch
(c) Occlusion experiment output

Figure 1: (a) Using a CNN to predict diabetic retinopathy in an OCT image/scan. (b) Occluding a part of the image changes the prediction probability. (c) By moving the occluding patch, a sensitivity heatmap can be produced.

Despite their successes, a key criticism of CNNs is that their internal workings are unintuitive to non-technical users. Thus, users often seek an "explanation" for why a CNN predicted a certain label. Explanations can help users trust CNNs, especially in high stakes applications such as radiology [10], and are a legal requirement for machine learning applications in some countries [27]. How to explain a CNN prediction is still an active research question, but in the practical literature, an already popular mechanism for CNN explanations is a simple procedure called *occlusion-based explanations* [29], or OBE for short.

OBE works as follows. Place a small patch (usually gray) on the image to occlude those pixels. Rerun CNN inference, illustrated in Figure 1(b), on the occluded image. The probability of the predicted class will change. Repeat this process by moving the patch across the image to obtain a sensitivity *heatmap* of probability changes, as Figure 1(c) shows. This heatmap highlights regions of the image that were highly "responsible" for the prediction (red/orange color regions). Such localization of the regions of interest allows users to gain intuition on what "mattered" for the prediction. For instance, the heatmap can highlight the diseased areas of a tissue image, which a radiologist can then inspect more for further tests. Overall, OBE is popular because it is easy for non-technical users to understand.

However, OBE is highly computationally expensive. Deep CNN inference is already expensive; OBE just amplifies it by issuing a large number of CNN re-inference requests (even thousands). For example, [31] reports 500,000 re-inference requests for 1 image, taking 1 hour even on a GPU! Such long wait times can hinder users' ability to consume explanations and reduce their productivity. One could use more compute hardware, if available, since OBE is embarrassingly parallel across re-inference requests. However, this may not always be affordable, especially for domain scientists, or feasible in all settings, e.g., in mobile clinical diagnosis. Extra hardware can also raise monetary costs, especially in the cloud.

To mitigate the above issue, we use a database-inspired lens to formalize and accelerate OBE. We start with a simple but crucial observation: *occluded images are not disjoint but share most of their pixels; so, most of the re-inference computations are redundant.* This observation leads us to connect OBE with two classical data management concerns: *incremental view maintenance* (IVM) and *multi-query optimization* (MQO). Instead of treating a CNN as a "black-box," we open it up and formalize *CNN layers* as "queries." Just like how a relational query converts relations to other relations, a CNN layer converts *tensors* (multidimensional arrays) to other tensors. A deep CNN stacks many types of such layers to convert the input (represented as a tensor) to the prediction output, as Figure 1(a) illustrates. So, we reimagine OBE as *a set of tensor transformation queries* with incrementally updated inputs. With this fresh database-inspired view, we devise several *novel CNN-specific query optimization techniques* to accelerate OBE.

Our first optimization is *incremental inference.* We first *materialize* all tensors produced by the CNN. For every re-inference request, instead of rerunning inference from scratch, we treat it as an IVM query, with the "views" being the tensors. We rewrite such queries to *reuse* the materialized views as much as possible and recompute only what is needed, thus *avoiding computational redundancy.* Such rewrites are non-trivial because they are tied to the complex geometric dataflows of CNN layers. We formalize such dataflows to create a novel *algebraic rewrite framework.* We also create a "static analysis" routine to tell up front how much computations can be saved. Going further, we batch all re-inference requests to reuse the *same* materialized views. This is a form of MQO we call *batched incremental inference.* We create a GPU-optimized kernel for such execution. To the best of our knowledge, this is the first time IVM is combined with MQO in query optimization, at least in machine learning (ML) systems.

We then introduce two novel *approximate inference* optimizations that allow users to tolerate some degradation in visual quality of the heatmaps produced to reduce runtimes further. These optimizations build upon our incremental inference optimization and use our IVM framework. Our first approximate optimization, *projective field thresholding*, draws upon an idea from neuroscience and exploits the internal semantics of how CNNs work. Our second, *adaptive drill-down*, exploits the semantics of the OBE task and the way users typically consume the heatmaps produced. We also present intuitive automated parameter tuning methods to help users adopt these optimizations. Our optimizations operate largely at the logical level and are complementary to more physical-level optimizations such as low-precision computation and model pruning.

We prototype our ideas in the popular deep learning framework PyTorch to create a tool we call Krypton. It works on both CPU and GPU. We perform an empirical evaluation of Krypton with multiple CNNs and real-world image datasets from recent radiology and ML papers. Krypton yields up to 35x speedups over the current dominant practice of running re-inference with just batching for producing high-quality approximate heatmaps, and up to 5x speedups for producing exact heatmaps.

This paper is a shortened version of our paper titled "Incremental and Approximate Inference for Faster Occlusion-based Deep CNN Explanations" that appeared in ACM SIGMOD 2019 [20]. More details about the techniques discussed in this paper and more experimental results can be found in that SIGMOD paper, as well as in the associated extended version published in ACM TODS [21].

2. SETUP AND PRELIMINARIES

We now state our problem formally and explain our assumptions. We then formalize the dataflow of the layers of a CNN, since these are required for understanding our techniques in Sections 3 and 4. Table 1 lists our notation.

2.1 Problem Statement and Assumptions

We are given a CNN f that has a sequence (or DAG) of *layers* l, each of which has a *tensor transformation function* $T_{:l}$. We are also given the image $\mathcal{I}_{:img}$ for which the occlusion-based explanation (OBE) is desired, the class label L predicted by f on $\mathcal{I}_{:img}$, an occlusion patch $\mathcal{P}$ in RGB format, and occlusion patch *stride* $S_{\mathcal{P}}$. We are also given a set of patch positions G constructed either automatically or manually with a visual interface interactively. The OBE workload is as follows: produce a 2-D heatmap M, wherein each value corresponds to a position in G and has the prediction probability of L by f on the occluded image $\mathcal{I}'_{x,y:img}$ (i.e., superimpose occlusion patch on image) or zero otherwise. More precisely, we can describe the OBE workload with the following logical statements:

$$W_M = \lfloor (\texttt{width}(\mathcal{I}_{:img}) - \texttt{width}(\mathcal{P}) + 1)/S_{\mathcal{P}} \rfloor \quad (1)$$

$$H_M = \lfloor (\texttt{height}(\mathcal{I}_{:img}) - \texttt{height}(\mathcal{P}) + 1)/S_{\mathcal{P}} \rfloor \quad (2)$$

$$M \in \mathbb{R}^{H_M \times W_M} \quad (3)$$

$$\forall\ (x,y) \in G : \quad (4)$$

$$\mathcal{I}'_{x,y:img} \leftarrow \mathcal{I}_{:img}\ \circ_{(x,y)}\ \mathcal{P} \quad (5)$$

$$M[x,y] \leftarrow f(\mathcal{I}'_{x,y:img})[L] \quad (6)$$

Steps (1) and (2) calculate the dimensions of the heatmap M. Step (5) superimposes $\mathcal{P}$ on $\mathcal{I}_{:img}$ with its top left corner placed on the (x,y) location of $\mathcal{I}_{:img}$. Step (6) calculates the output value at the (x,y) location by performing CNN inference for $\mathcal{I}'_{x,y:img}$ using f and picks the prediction probability of L. Steps (5) and (6) are performed *independently* for every position in G. In the *non-interactive* mode, G is initialized to $G = [0, H_M) \times [0, W_M)$. Intuitively, this represents the set of all possible occlusion patch positions on $\mathcal{I}_{:img}$, which yields a full heatmap. In the *interactive* mode, the user manually places the occlusion patch only at a few locations at a time, yielding partial heatmaps.

2.2 Dataflow of CNN Layers

CNNs are organized as *layers* of various types, each of which transforms a tensor (multidimensional array, typically 3-D) into another tensor: *Convolution* uses image

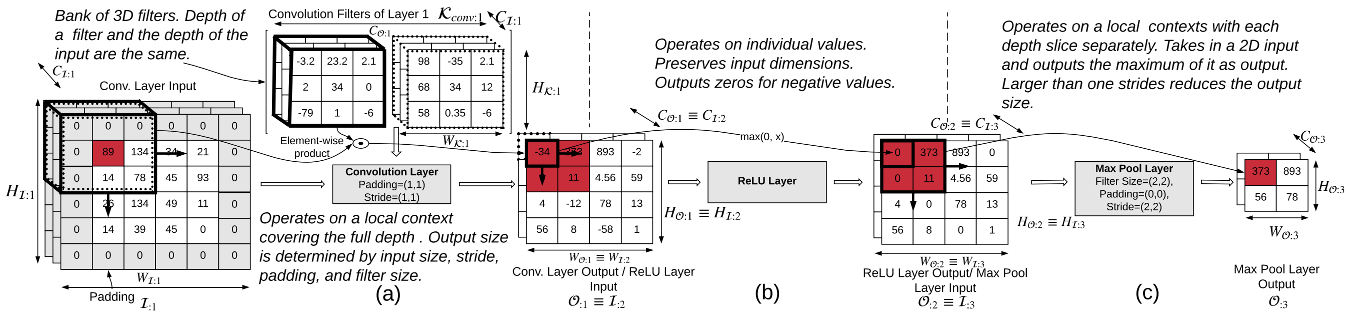

Bank of 3D filters. Depth of a filter and the depth of the input are the same.
Convolution Filters of Layer 1 $\mathcal{K}_{conv:1}$
$C_{\mathcal{I}:1}$
Conv. Layer Input
$H_{\mathcal{I}:1}$
$W_{\mathcal{I}:1}$
Padding $\mathcal{I}_{:1}$
Element-wise product
Convolution Layer Padding=(1,1) Stride=(1,1)
Operates on a local context covering the full depth . Output size is determined by input size, stride, padding, and filter size.
(a)
$H_{\mathcal{K}:1}$
$W_{\mathcal{K}:1}$
$C_{\mathcal{O}:1}$
$C_{\mathcal{O}:1} \equiv C_{\mathcal{I}:2}$
$H_{\mathcal{O}:1} \equiv H_{\mathcal{I}:2}$
$W_{\mathcal{O}:1} \equiv W_{\mathcal{I}:2}$
Conv. Layer Output / ReLU Layer Input
$\mathcal{O}_{:1} \equiv \mathcal{I}_{:2}$
Operates on individual values. Preserves input dimensions. Outputs zeros for negative values.
max(0, x)
ReLU Layer
(b)
$C_{\mathcal{O}:2} \equiv C_{\mathcal{I}:3}$
$H_{\mathcal{O}:2} \equiv H_{\mathcal{I}:3}$
$W_{\mathcal{O}:2} \equiv W_{\mathcal{I}:3}$
ReLU Layer Output/ Max Pool Layer Input
$\mathcal{O}_{:2} \equiv \mathcal{I}_{:3}$
Operates on a local contexts with each depth slice separately. Takes in a 2D input and outputs the maximum of it as output. Larger than one strides reduces the output size.
Max Pool Layer Filter Size=(2,2), Padding=(0,0), Stride=(2,2)
(c)
$C_{\mathcal{O}:3}$
$H_{\mathcal{O}:3}$
$W_{\mathcal{O}:3}$
Max Pool Layer Output
$\mathcal{O}_{:3}$

Figure 2: Simplified illustration of the key layers of a typical CNN. The highlighted cells (dark/red background) show how a small local spatial context in the first input propagates through subsequent layers. (a) Convolution layer (for simplicity sake, bias addition is not shown). (b) ReLU Non-linearity layer. (c) Pooling layer (max pooling). Notation is explained in Table 1.

Symbol	Meaning
f	Given deep CNN; input is an image tensor; output is a probability distribution over class labels
L	Class label predicted by f for the original image $\mathcal{I}_{:img}$
$T_{:l}$	Tensor transformation function of layer l of the given CNN f
$\mathcal{P}$	Occlusion patch in RGB format
$S_{\mathcal{P}}$	Occlusion patch striding amount
G	Set of occlusion patch superimposition positions on $\mathcal{I}_{:img}$ in (x,y) format
M	Heatmap produced by the OBE workload
H_M, W_M	Height and width of M
$\circ_{(x,y)}$	Superimposition operator. $A \circ_{(x,y)} B$, superimposes B on top of A starting at (x, y) position
$\mathcal{I}_{:l}$ ($\mathcal{I}_{:img}$)	Input tensor of layer l (Input Image)
$\mathcal{O}_{:l}$	Output tensor of layer l
$C_{\mathcal{I}:l}, H_{\mathcal{I}:l}, W_{\mathcal{I}:l}$	Depth, height, and width of input of layer l
$C_{\mathcal{O}:l}, H_{\mathcal{O}:l}, W_{\mathcal{O}:l}$	Depth, height, and width of output of layer l
$\mathcal{K}_{conv:l}$	Convolution filter kernels of layer l
$\mathcal{B}_{conv:l}$	Convolution bias value vector of layer l
$\mathcal{K}_{pool:l}$	Pooling filter kernel of layer l
$H_{\mathcal{K}:l}, W_{\mathcal{K}:l}$	Height and width of filter kernel of layer l
$S_{:l}; S_{x:l}; S_{y:l}$	Filter kernel striding amounts of layer l; $S_{:l} \equiv (S_{x:l}, S_{y:l})$, strides along width and height dimensions
$P_{:l}; P_{x:l}; P_{y:l}$	Padding amounts of layer l; $P_{:l} \equiv (P_{x:l}, P_{y:l})$, padding along width and height dimensions

Table 1: Notation used in this paper.

filters from graphics to extract features, but with parametric filter weights (learned during training); *Pooling* subsamples features in a spatial-aware manner; *Batch-Normalization* normalizes the output tensor; *Non-Linearity* applies an element-wise non-linear function (e.g., ReLU); *Fully-Connected* is an ordered collection of perceptrons [9]. The output tensor of a layer can have a different width, height, and/or depth than the input. An image can be viewed as a tensor, e.g., a 224×224 RGB image is a 3-D tensor with width and height 224 and depth 3. A Fully-Connected layer converts a 1-D tensor (or a "flattened" 3-D tensor) to another 1-D tensor. For simplicity of exposition, we group CNN layers into 3 main categories based on the *spatial locality* of how they transform a tensor: (1) Transformations with a *global context*; (2) Transformations at the granularity of *individual elements*; and (3) Transformations at the granularity of a *local spatial context.*

Global context granularity. Such layers convert the input tensor into an output tensor using one global transformation. Since, every element of the output will likely be affected by a point change in the input, such layers do not offer a major opportunity for incremental computations. Fully-Connected is the only layer of this type. They typically arise only as the last layer(s) in deep CNNs (and never in some recent deep CNNs), and typically account for a negligible fraction of the total computational cost.

Individual element granularity. Such layers apply a "map()" function on the elements of the input tensor, as Figure 2 (b) illustrates. Non-Linearity (e.g., ReLU) falls under this category. If the input is incrementally updated, only the corresponding region of the output will be affected. Thus, incremental inference for such layers is straightforward.

Local spatial context granularity. Such layers perform weighted aggregations of slices of the input tensor, called *local spatial contexts*, by multiplying them with a *filter kernel* (a tensor of weights). If the input is incrementally updated, the region of the output that will be affected is not straightforward to ascertain–this requires non-trivial and careful calculations due to the overlapping nature of how filters get applied to local spatial contexts. Both Convolution and Pooling fall under this category. Since such layers typically account for the bulk of the computational cost of deep CNN inference, enabling incremental inference for such layers in the OBE context is a key focus of this paper (Section 3). The rest of this section explains the machinery of the dataflow in such layers using our notation.

Dataflow of Convolution Layers. A layer l has $C_{\mathcal{O}:l}$ 3-D filter kernels arranged as a 4-D array $\mathcal{K}_{conv:l}$, with each having a smaller spatial width $W_{\mathcal{K}:l}$ and height $H_{\mathcal{K}:l}$ than the width $W_{\mathcal{I}:l}$ and height $H_{\mathcal{I}:l}$ of the input tensor $\mathcal{I}_{:l}$ but the same depth $C_{\mathcal{I}:l}$. During inference, c^{th} filter kernel is "strided" along the width and height dimensions of the input to produce a 2-D "activation map" $A_{:c} = (a_{y,x:c}) \in \mathbb{R}^{H_{\mathcal{O}:l} \times W_{\mathcal{O}:l}}$ by computing element-wise products between the kernel and the local spatial context and adding a bias value. The computational cost of each of these small matrix products is proportional to the volume of the filter kernel. All the 2-D activation maps are then stacked along the depth dimension to produce the output tensor $\mathcal{O}_{:l} \in \mathbb{R}^{C_{\mathcal{O}:l} \times H_{\mathcal{O}:l} \times W_{\mathcal{O}:l}}$. Figure 2 (a) presents a simplified illustration of this layer.

Dataflow of Pooling Layers. Such layers behave essentially like Convolution layers with a fixed (not learned) 2-D filter kernel $\mathcal{K}_{pool:l}$. These kernels aggregate a local spatial context to compute its maximum or average element. However, unlike Convolution, Pooling operates on the depth slices of the input tensor independently. Figure 2(c) presents a simplified illustration of this layer. Since OBE only concerns the width and height dimensions of the image and subsequent tensors, we treat both these types of layers in a unified manner for our optimizations.

Relationship between Input and Output Dimensions. For Convolution and Pooling layers, $W_{\mathcal{O}:l}$ and $H_{\mathcal{O}:l}$ are determined by $W_{\mathcal{I}:l}$ and $H_{\mathcal{I}:l}$, $W_{\mathcal{K}:l}$ and $H_{\mathcal{K}:l}$, and two other parameters that are specific to that layer: *stride* $S_{:l}$ and *padding* $P_{:l}$. Stride is the number of pixels by which the filter kernel is moved at a time. For some layers, to help control the dimensions of the output to be the same as the input, one "pads" the input with zeros. *Padding* $P_{:l}$ captures how much such padding extends these dimensions. Both stride and padding values can differ along the width and height dimensions; $S_{x:l}$ and $S_{y:l}$ and $P_{x:l}$ and $P_{y:l}$, respectively. In Figure 2, the Convolution layer has $S_{x:l} = S_{y:l} = 1$, while the Pooling layer has $S_{x:l} = S_{y:l} = 2$. Convolution layer also has $P_{x:l} = P_{y:l} = 1$. Given these parameters, width (similarly height) of the output tensor is given by the following formula:

$$W_{\mathcal{O}:l} = (W_{\mathcal{I}:l} - W_{\mathcal{K}:l} + 1 + 2 \times P_{x:l})/S_{x:l} \quad (7)$$

Computational Cost of Inference. Convolution layers typically account for a bulk of the cost (90% or more). Thus, we can roughly estimate the computational cost of inference by counting the number of *fused multiply-add* (FMA) floating point operations (FLOPs) needed for the Convolution layers. The amount of computations performed by a single application of a Convolution filter kernel $\mathcal{K}_{:l}$ is equal to the volume of the filter in FLOPs, with each FLOP corresponding to one FMA. Thus, the total computational cost $Q_{:l}$ of a layer that produces output $\mathcal{O}_{:l}$ and the total computational cost Q of the entire set of Convolution layers of a given CNN f can be calculated as per Equations (8) and (9).

$$Q_{:l} = (C_{\mathcal{I}:l} \cdot H_{\mathcal{K}:l} \cdot W_{\mathcal{K}:l})(C_{\mathcal{O}:l} \cdot H_{\mathcal{O}:l} \cdot W_{\mathcal{O}:l}) \quad (8)$$

$$Q = \sum_{l \ in \ f} Q_{:l} \quad (9)$$

3. INCREMENTAL CNN INFERENCE

In relational IVM, when a part of the input relation is updated, we recompute only the part of the output that changes. We bring this notion to CNNs; a CNN layer is our "query" and a materialized feature tensor is our "relation." OBE updates only a part of the image. So, only some parts of the tensors need to be recomputed. We call this *incremental inference*. We create an algebraic framework to determine which parts of a CNN layer must be updated and how to propagate updates across layers. We then combine our incremental inference framework with an MQO-style technique and characterize theoretical upper bounds on the speedups possible with these ideas.

3.1 Single Layer Incremental Inference

As per the discussion in Section 2.2, we focus only on the non-trivial layers that operate at the granularity of a local spatial context (Convolution and Pooling). Table 2 lists some extra notation for this section.

Symbol	Meaning
$x^{\mathcal{I}}_{\mathcal{P}:l}, y^{\mathcal{I}}_{\mathcal{P}:l}$	Start coordinates of input update patch for layer l
$x^{\mathcal{R}}_{\mathcal{P}:l}, y^{\mathcal{R}}_{\mathcal{P}:l}$	Start coordinates of read-in context for layer l
$x^{\mathcal{O}}_{\mathcal{P}:l}, y^{\mathcal{O}}_{\mathcal{P}:l}$	Start coordinates of output update patch for layer l
$H^{\mathcal{I}}_{\mathcal{P}:l}, W^{\mathcal{I}}_{\mathcal{P}:l}$	Height and width of input update patch for layer l
$H^{\mathcal{R}}_{\mathcal{P}:l}, W^{\mathcal{R}}_{\mathcal{P}:l}$	Height and width of read-in context for layer l
$H^{\mathcal{O}}_{\mathcal{P}:l}, W^{\mathcal{O}}_{\mathcal{P}:l}$	Height and width of output update patch for layer l
τ	Projective field threshold
$r_{drill-down}$	Drill-down fraction for adaptive drill-down

Table 2: Additional notation for Sections 3 and 4.

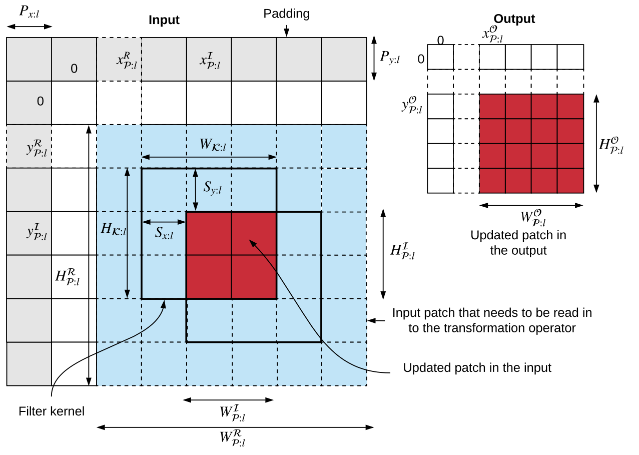

$P_{x:l}$
Input
Padding
Output
0
$x^{\mathcal{R}}_{\mathcal{P}:l}$
$x^{\mathcal{I}}_{\mathcal{P}:l}$
$P_{y:l}$
$x^{\mathcal{O}}_{\mathcal{P}:l}$
$y^{\mathcal{R}}_{\mathcal{P}:l}$
$W_{\mathcal{K}:l}$
$S_{y:l}$
$y^{\mathcal{I}}_{\mathcal{P}:l}$
$H_{\mathcal{K}:l}$
$S_{x:l}$
$H^{\mathcal{R}}_{\mathcal{P}:l}$
$H^{\mathcal{I}}_{\mathcal{P}:l}$
$y^{\mathcal{O}}_{\mathcal{P}:l}$
$H^{\mathcal{O}}_{\mathcal{P}:l}$
$W^{\mathcal{O}}_{\mathcal{P}:l}$
Updated patch in the output
Input patch that needs to be read in to the transformation operator
Updated patch in the input
Filter kernel
$W^{\mathcal{I}}_{\mathcal{P}:l}$
$W^{\mathcal{R}}_{\mathcal{P}:l}$

Figure 3: Simplified illustration of input and output update patches for Convolution/Pooling layers.

Determining Patch Update Locations. We first explain how to calculate the coordinates and dimensions of the *output update patch* of layer l given the *input update patch* and layer-specific parameters. Figure 3 illustrates these calculations. Our coordinate system's origin is at the top left corner. The input update patch is shown in red/dark color and starts at $(x^{\mathcal{I}}_{\mathcal{P}:l}, y^{\mathcal{I}}_{\mathcal{P}:l})$, with height $H^{\mathcal{I}}_{\mathcal{P}:l}$ and width $W^{\mathcal{I}}_{\mathcal{P}:l}$. The output update patch starts at $(x^{\mathcal{O}}_{\mathcal{P}:l}, y^{\mathcal{O}}_{\mathcal{P}:l})$ and has a height $H^{\mathcal{O}}_{\mathcal{P}:l}$ and width $W^{\mathcal{O}}_{\mathcal{P}:l}$. Due to overlaps among filter kernel positions during inference, computing the output update patch requires reading a slightly larger spatial context than the input update patch–we call this the "read-in context," and it is illustrated by the blue/shaded region in Figure 3. The read-in context starts at $(x^{\mathcal{R}}_{\mathcal{P}:l}, y^{\mathcal{R}}_{\mathcal{P}:l})$, with its dimensions denoted by $W^{\mathcal{R}}_{\mathcal{P}:l}$ and $H^{\mathcal{R}}_{\mathcal{P}:l}$. The relationship between these quantities along the width dimension can be expressed as follows (likewise for the height dimension):

$$x^{\mathcal{O}}_{\mathcal{P}:l} = max\big(\lceil (P_{x:l} + x^{\mathcal{I}}_{\mathcal{P}:l} - W_{\mathcal{K}:l} + 1)/S_{x:l} \rceil, 0\big) \quad (10)$$

$$W^{\mathcal{O}}_{\mathcal{P}:l} = min\big(\lceil (W^{\mathcal{I}}_{\mathcal{P}:l} + W_{\mathcal{K}:l} - 1)/S_{x:l} \rceil, W_{\mathcal{O}:l}\big) \quad (11)$$

$$x^{\mathcal{R}}_{\mathcal{P}:l} = x^{\mathcal{O}}_{\mathcal{P}:l} \times S_{x:l} - P_{x:l} \quad (12)$$

$$W^{\mathcal{R}}_{\mathcal{P}:l} = W_{\mathcal{K}:l} + (W^{\mathcal{O}}_{\mathcal{P}:l} - 1) \times S_{x:l} \quad (13)$$

Equation (10) calculates the coordinates of the output update patch. As shown in Figure 3, padding effectively shifts the coordinate system and thus, $P_{x:l}$ is added to correct it. Due to overlaps among the filter kernels, the affected region of the input update patch (blue/shaded region in Figure 3) will be increased by $W_{\mathcal{K}:l} - 1$, which needs to be subtracted from the input coordinate $x^{\mathcal{I}}_{\mathcal{P}:l}$. A filter of size $W_{\mathcal{K}:l}$ that is placed starting at $x^{\mathcal{I}}_{\mathcal{P}:l} - W_{\mathcal{K}:l} + 1$ will see an update starting from $x^{\mathcal{I}}_{\mathcal{P}:l}$. Equation (11) calculates the width of the output update patch, which is essentially the number of filter kernel stride positions on the read-in input context. However, this value cannot be larger than the output size. Given these, a start coordinate and width of the read-in context are given by Equations (12) and (13); similar equations hold for the height dimension (skipped for brevity).

Incremental Inference Operation. For layer l, given the transformation function $T_{:l}$, the pre-materialized input tensor $\mathcal{I}_{:l}$, input update patch $\mathcal{P}^{\mathcal{O}}_{:l}$, and the above calculated coordinates and dimensions of the input, output, and read-in context, the output update patch $\mathcal{P}^{\mathcal{O}}_{:l}$ is computed as follows:

$$\mathcal{U} = \mathcal{I}_{:l}[:, x^{\mathcal{R}}_{\mathcal{P}:l} : x^{\mathcal{R}}_{\mathcal{P}:l} + W^{\mathcal{R}}_{\mathcal{P}:l}, y^{\mathcal{R}}_{\mathcal{P}:l} : y^{\mathcal{R}}_{\mathcal{P}:l} + H^{\mathcal{R}}_{\mathcal{P}:l}] \quad (14)$$

$$\mathcal{U} = \mathcal{U} \circ_{(x^{\mathcal{I}}_{\mathcal{P}:l} - x^{\mathcal{R}}_{\mathcal{P}:l}),(y^{\mathcal{I}}_{\mathcal{P}:l} - y^{\mathcal{R}}_{\mathcal{P}:l})} \mathcal{P}^{\mathcal{I}}_{:l} \quad (15)$$

$$\mathcal{P}^{\mathcal{O}}_{:l} = T_{:l}(\mathcal{U}) \quad (16)$$

Equation (14) slices the read-in context $\mathcal{U}$ from the pre-materialized input tensor $\mathcal{I}_{:l}$. Equation (15) superimposes the input update patch $\mathcal{P}^{\mathcal{I}}_{:l}$ on it. This is an in-place update of the array holding the read-in context. Finally, Equation (16) computes the output update patch $\mathcal{P}^{\mathcal{O}}_{:l}$ by invoking $T_{:l}$ on $\mathcal{U}$. Thus, we avoid performing inference on all of $\mathcal{I}_{:l}$, thus achieving incremental inference and reducing FLOPs.

3.2 Propagating Updates across Layers

Unlike relational IVM, CNNs have many layers, often in a sequence. This is analogous to a sequence of queries, each requiring IVM on its predecessor's output. This leads to a new issue: correctly and automatically configuring the update patches across layers of a CNN. While this seems simple, it requires care at the boundary of a local context transformation and a global context transformation. In particular, we need to materialize the full updated output, not just the output update patches, since global context transformations lose spatial locality for subsequent layers. Some recent deep CNNs have a more general directed acyclic graph (DAG) structure for layers. They have two new kinds of layers that "merge" two branches in the DAG: *element-wise addition* and *depth-wise concatenation.* To address such cases, we propose a simple unified solution: compute the *bounding box* of the input update patches. While this will potentially recompute parts of the output that do not get modified, we think this trade-off is acceptable because the gains are likely to be marginal for the additional complexity introduced.

3.3 Multi-Query Incremental Inference

OBE issues $|G|$ re-inference requests *in one go.* Viewing each request as a "query" makes the connection with MQO [26] clear. The $|G|$ queries are also *not disjoint*, as the occlusion patch is small, which means most pixels are the same. We now briefly explain how we extend our IVM framework with an MQO-style optimization fusing multiple re-inference requests. An analogy with relational queries is many concurrent incremental updates on the same relation.

Batched Incremental Inference. Our optimization works as follows: materialize all CNN tensors *once* and *reuse* them for incremental inference across all $|G|$ queries. Since the occluded images share most of their pixels, parts of the tensors will likely be identical too. Thus, we can amortize the materialization cost. Batched execution is standard practice on high-throughput compute hardware like GPUs, since it amortizes CNN set up costs, data movement costs, etc. Batch sizes are tuned to optimize hardware utilization. Thus, we combine both these ideas to execute incremental inference in a batched manner. We call this approach "batched incremental inference." Empirically, we found that batching alone yields limited speedups (under 2X), but batched incremental inference amplifies the speedups.

GPU Optimized Implementation. Empirically, we found a dichotomy between CPUs and GPUs: batched incremental inference yielded expected speedups on CPUs, but it performed dramatically poorly on GPUs. In fact, a naive implementation on GPUs was *slower* than full re-inference! The reason for this was the overheads incurred during read-in context preparation step, which throttles the GPU throughput. To overcome this issue, we created a custom CUDA kernel to perform read-in context preparation more efficiently by *copying memory regions in parallel* for all items in the batched inference request.

3.4 Expected Speedups

We extend our framework to perform "static analysis" on a given CNN f to find how much FLOPs can be saved using incremental inference, yielding us an upper bound on speedups. The computational cost of incremental inference for a layer is proportional to the volume of the individual filter kernel times the total volume of the updated output. The total computational cost for incremental inference, denoted Q_{inc}, is the sum of incremental inference cost across all layers. Q_{inc} can be much smaller than Q in Equation (9). We define the *theoretical speedup* as the ratio $\frac{Q}{Q_{inc}}$. This tells us how beneficial incremental inference can be in the best case *without* running the actual inference itself.

We calculated the theoretical speedups for many popular CNNs for occlusion patches with varying sizes placed at the center of the image. For an occlusion patch of size 16×16, VGG-16 sees the highest theoretical speedups of 6x; DenseNet-121 sees a speedup of 2x, the lowest. Most CNNs fall in the 2x–3x range. The differences arise due to the specifics of the CNNs' architectures: VGG-16 has small Convolution filter kernels and strides, which means full re-inference is costlier. While speedups of 2x–3x may sound "not that significant" in practice, we find they are indeed significant for two reasons. First, *users often wait in the loop* for OBE when performing interactive diagnoses. Thus, even such speedups can improve their productivity. Second, our IVM is the *foundation for our approximate inference* optimizations (Section 4), which amplify the speedups.

4. APPROXIMATE CNN INFERENCE

Since incremental inference is *exact*, i.e., it yields the same heatmap as full inference, it does not exploit a capability of human perception: tolerance of some degradation in visual quality. We now briefly explain how we build upon our IVM framework to create two novel heuristic approximate inference optimizations that trade off the heatmap's quality in a user-tunable manner to accelerate OBE further.

4.1 Projective Field Thresholding

The *projective field* of a CNN neuron is the slice of the output tensor that is connected to it. It is a term from neuroscience to describe the effects of a retinal cell on the output of the eye's neuronal circuitry [7]. This notion sheds light on the *growth of the size* of the update patches through the layers of a CNN. The 3 kinds of layers (Section 2.2) affect the projective field size growth differently. Individual element transformations do not alter the projective field size. Global context transformations increase it to the whole output. However, local spatial context transformations, which are the most crucial, increase it *gradually* at a rate determined by the filter kernel's size and stride: additively in the size and multiplicatively in the stride. The growth of the projective field size implies the amount of FLOPs saved by IVM decreases as we go to the higher layers of a CNN. Eventually, the output update patch becomes as large as the output tensor. This growth is illustrated by Figure 4(a).

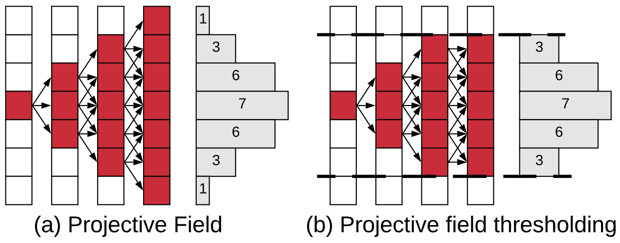


Figure 4: (a) Projective field growth for 1-D Convolution (filter size 2, stride 1). (b) Projective field *thresholding*; $\tau = 5/7$.

Our above observation motivates the main idea of this optimization, which we call projective field thresholding: *truncate* the projective field from growing beyond a given *threshold fraction* τ ($0 < \tau \leq 1$) of the output size. This means inference in subsequent layers is approximate. Figure 4(b) illustrates the idea for a filter size 3 and stride 1. This approximation can alter the accuracy of the output values and the heatmap's visual quality. Empirically, we find that modest truncation is tolerable and does not affect the heatmap's visual quality too significantly.

To provide intuition on why the above happens, consider histograms shown in Figures 4(a,b) that list the number of unique "paths" from the updated element to each output element. It resembles a Gaussian distribution. Thus, for most of the output patch updates, truncation will only discard a few values at the "fringes" that contribute to an output element. This optimization is only feasible *in conjunction with* our incremental inference framework (Section 3) to reuse the remaining parts of the tensors and save FLOPs.

4.2 Adaptive Drill-Down

This heuristic optimization is based on our observation about a peculiar semantics of OBE that lets us modify how G (the set of occlusion patch locations) is specified and handled, especially in the non-interactive specification mode. We explain our intuition with an example. Consider a radiologist explaining a CNN prediction for diabetic retinopathy on a tissue image. The region of interest typically occupies only a tiny fraction of the image. Thus, it is not necessary to perform regular OBE for *every* patch location: most of the (incremental) inference computations are effectively "wasted" on uninteresting regions. In such cases, we modify the OBE workflow to produce an approximate heatmap using a two-stage process, illustrated by Figure 5.

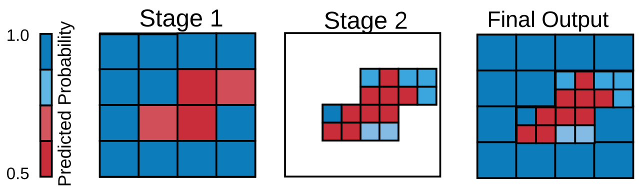


Figure 5: Schematic representation of *adaptive drill-down.*

In stage one, we produce a lower resolution heatmap by using a larger stride–we call it *stage one stride* S_1. Using this heatmap, we identify the regions of the input that see the largest drops in predicted probability for label L. Given a predefined parameter *drill-down fraction*, denoted $r_{drill-down}$, we select a proportional number of regions based on the probability drops. In stage two, we perform OBE only for these regions with original stride value (we call this *stage two stride*, S_2) to yield a portion of the heatmap at the original higher resolution. This optimization also builds upon our incremental inference optimizations, but it is *orthogonal* to projective field thresholding.

4.3 Automated Parameter Tuning

We also devise automated parameter tuning methods for easily configuring the approximate inference optimizations. For projective field thresholding, mapping a threshold value (τ) to visual quality directly is likely to be unintuitive for users. Thus, to measure visual quality more intuitively, we adopt a cognitive science-inspired metric called Structural Similarity (SSIM) Index, which is widely used to quantify human-perceptible differences between two images [28]. During an offline phase, we learn a function that maps the heatmap visual quality to a τ value using a sample of workload images. During the online phase, we use the learned function to map the user given SSIM value to a target τ value. For adaptive drill-down, we expect the user to provide the drill-down ratio ($r_{drill-down}$) based on her understanding of the size of the region of interest in the OBE heatmap and on how much speedup she wants to achieve. We set the stage one stride (S_1) using these two user-given settings.

5. EXPERIMENTAL EVALUATION

We integrated our techniques with the popular deep learning tool PyTorch to create a system we call KRYPTON. We now present a snapshot of our key empirical results with KRYPTON on different CNNs and datasets.

Datasets. We use 2 real-world image datasets: *OCT* and *Chest X-Ray*. *OCT* has about 84,000 optical coherence tomography retinal images with 4 classes. *Chest X-Ray* has about 6,000 X-ray images with 3 classes. Both *OCT* and *Chest X-Ray* are from a recent radiology study that applied deep CNNs to detect the respective diseases [11].

Workloads. We use 3 diverse ImageNet-trained [25] deep CNNs: VGG16, ResNet18 and Inception3. They comple-

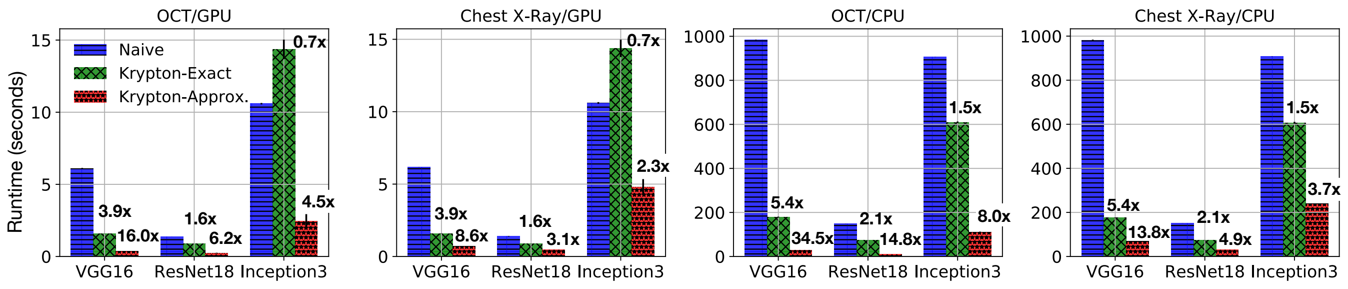


Figure 6: End-to-end runtimes of Krypton and the baseline on 2 datasets and 3 CNNs on GPU and CPU.

ment each other in terms of model size, architectural complexity, computational cost, and our predicted theoretical speedups. CNNs were fine-tuned by retraining their final Fully-Connected layers using the *OCT* and *Chest X-Ray* datasets, as per standard practice. The GPU-based experiments used a batch size of 128; for CPUs, the batch size was 16. All CPU-based experiments were executed with a thread parallelism of 8.

Experimental Setup. We use a machine with 32 GB RAM, Intel i7 3.4GHz CPU, and NVIDIA Titan X (Pascal) GPU with 12 GB memory. The machine runs Ubuntu 16.04 with PyTorch version 0.4.0, CUDA version 9.0, and cuDNN version 7.1.2. All reported runtimes are the average of 3 runs, with 95% confidence intervals shown.

5.1 End-to- End Runtimes

We focus on perhaps the most common scenario for OBE: produce the whole heatmap for automatically created G ("non-interactive" mode). The occlusion patch size is set to 16; stride, 4. We compare two variants of Krypton: Krypton-Exact uses only incremental inference, while Krypton-Approximate uses our approximate inference optimizations too. The baseline is *Naive*, which runs full re-inference with only batching to improve hardware utilization. We set the approximate inference parameters based on the semantics of each dataset's prediction task. Figure 6 presents the results. More details about the parameters and visual examples of the heatmaps are available in the longer version of this paper [20].

Overall, we see that Krypton offers significant speedups across the board on both GPU and CPU, with the highest speedups seen by Krypton-Approximate on *OCT* with VGG16: 16x on GPU and 34.5x on CPU. The highest speedups of Krypton-Exact are also on VGG16: 3.9x on GPU and 5.4x on CPU. The speedups of Krypton-Exact are identical across datasets for a given CNN, since it does not depend on the image semantics, unlike Krypton-Approximate due to its parameters. Krypton-Approximate sees the highest speedups on *OCT*.

The speedups are lower with ResNet18 and Inception3 than VGG16 due to their architectural properties (kernel filter dimensions, stride, etc.) that make the projective field grow faster. Moreover, Inception3 has a complex DAG architecture with more branches and depth-wise concatenation, which limits GPU throughput for incremental inference. In fact, Krypton-Exact on GPU shows a minor slow-down (0.7x) with Inception3. However, Krypton-Approximate still offers speedups on GPU with Inception3 (up to 4.5x). We also found that ResNet18 and VGG16 see speedups almost near their theoretical speedups, but Inception3 does not. Note that our theoretical speedup definition only counts FLOPs and does not account for memory stall overheads.

Finally, the speedups are higher on CPU than GPU; this is because CPU suffers less from memory stalls during incremental inferences. However, the *absolute* runtimes are much lower on GPU, as expected. Overall, Krypton reduces OBE runtimes substantially for multiple datasets and deep CNNs.

5.2 Other Experimental Results

We also perform ablation studies to evaluate the impact of each of our optimization techniques for varying configuration parameters for OBE. The patch size and stride have an inverse effect on speedups because they reduce the sheer amount of FLOPs in the re-inference requests. The parameters of the approximate optimizations also affect speedups significantly, and our automated tuning methods help optimize the accuracy-runtime tradeoffs effectively. The memory overhead of our batched incremental inference approach is also significantly lower (about 2x) compared to full re-inference.

5.3 Demonstration and Extensions

In follow-on work, we extended Krypton and demonstrated support for human-in-the-loop OBE [19, 24]. The user can interactively select a sub-region of the image (to specify G) and iteratively refine it. We also showed that Krypton can help accelerate OBE on time-series data out of the box and can also help accelerate object recognition in fixed-angle camera videos when combined with new approximate inference techniques [21].

6. OTHER RELATED WORK

Explaining CNN Predictions. Perturbation-based and gradient-based are the two main kinds of methods for explaining CNN predictions. Perturbation-based methods observe the output of the CNN by modifying regions of the input image. OBE belongs to this category. In practice, however, OBE is usually more popular among domain scientific users, especially in radiology [10], since it is easy to understand for non-technical users and typically produces high-quality heatmaps.

Faster CNN Inference. EVA^2 [3] and CBInfer [4] use approximate change detection for faster CNN inference over video data. While one can map OBE to a "video," our IVM and MQO techniques are complementary to such systems, while our approximate inference optimizations are also novel and exploit specific properties of CNNs and OBE.

Query Optimization. Our work is inspired by the long line of work on relational IVM [6, 16], but ours is the first to use the IVM lens for OBE with CNNs. Our algebraic IVM framework is closely tied to the dataflow of CNN layers, which transform tensors in non-trivial ways. Our work is related to the IVM framework for linear algebra in [23]. They focus on bulk matrix operators and incremental addition of rows. The focus of our work is on more fine-grained CNN inference computations. Our work is also inspired by relational MQO [26], but our focus is CNN inference, not relational queries. MQO for ML systems is a growing area of research [2, 14, 15], both for classical statistical ML (e.g., [5, 12, 13, 17, 30]) and deep learning (e.g., [18, 22]). Our work adds to this direction, but ours is the first work to combine MQO with IVM for ML systems. Our approximate inference optimizations are inspired by AQP [8], but unlike statistical approximations of SQL aggregates, our techniques are novel CNN-specific and human perception-aware heuristics tailored to OBE.

7. CONCLUSIONS AND FUTURE WORK

Deep CNNs are popular for image prediction tasks, but their internal workings are unintuitive for most users. Occlusion-based explanation (OBE) is a popular mechanism to explain CNN predictions, but it is highly compute-intensive. We formalize OBE from a data management standpoint and present several novel database-inspired optimizations to speed up OBE. Our techniques span incremental inference and multi-query optimization for CNNs to human perception-aware approximate inference. Overall, our ideas yield over an order of magnitude speedups for OBE on both GPU and CPU. As for future work, we plan to extend our ideas to other deep learning workloads and data types. More broadly, we believe database-inspired query optimization techniques can help reduce resource costs of deep learning systems significantly, thus enabling a wider base of application users to benefit from modern ML.

Acknowledgments. This work was supported in part by a Hellman Fellowship and by the NIDDK of the NIH under award number R01DK114945. The content is solely the responsibility of the authors and does not necessarily represent the official views of the NIH. We thank NVIDIA Corporation for donating the Titan Xp GPU used for this work. We thank the members of UC San Diego's Database Lab for their feedback on this work.

8. REFERENCES

[1] AI Device for Detecting Diabetic Retinopathy Earns Swift FDA Approval. `https://bit.ly/36300H9`. Accessed April 30, 2020.
[2] M. Boehm et al. *Data Management in Machine Learning Systems.* Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2019.
[3] M. Buckler et al. EVA^2: Exploiting Temporal Redundancy in Live Computer Vision. In *ISCA*, 2018.
[4] L. Cavigelli et al. CBInfer: Change-based Inference for Convolutional Neural Networks on Video Data. In *International Conference on Distributed Smart Cameras*, 2017.
[5] L. Chen et al. Towards Linear Algebra over Normalized Data. In *VLDB*, 2017.
[6] R. Chirkova and J. Yang. *Materialized Views.* Now Publishers Inc., 2012.
[7] S. E. de Vries et al. The Projective Field of a Retinal Amacrine Cell. In *Journal of Neuroscience*, 2011.
[8] M. N. Garofalakis and P. B. Gibbons. Approximate Query Processing: Taming the TeraBytes. In *VLDB*, 2001.
[9] I. Goodfellow et al. *Deep Learning.* MIT press Cambridge, 2016.
[10] K.-H. Jung et al. Deep Learning for Medical Image Analysis: Applications to Computed Tomography and Magnetic Resonance Imaging. *Hanyang Medical Reviews*, 2017.
[11] D. S. Kermany et al. Identifying Medical Diagnoses and Treatable Diseases by Image-Based Deep Learning. *Cell*, 2018.
[12] P. Konda et al. Feature Selection in Enterprise Analytics: A Demonstration Using an R-Based Data Analytics System. In *VLDB*, 2013.
[13] A. Kumar et al. Learning Generalized Linear Models Over Normalized Data. In *ACM SIGMOD*, 2015.
[14] A. Kumar et al. Model Selection Management Systems: The Next Frontier of Advanced Analytics. *ACM SIGMOD Rec.*, 2016.
[15] A. Kumar et al. Data Management in Machine Learning: Challenges, Techniques, and Systems. In *ACM SIGMOD*, 2017.
[16] A. Y. Levy et al. Answering Queries Using Views. In *PODS*, 1995.
[17] S. Li et al. Enabling and Optimizing Non-Linear Feature Interactions in Factorized Linear Algebra. In *ACM SIGMOD*, 2019.
[18] S. Nakandala et al. Cerebro: Efficient and Reproducible Model Selection on Deep Learning Systems. In *ACM SIGMOD DEEM Workshop*, 2019.
[19] S. Nakandala et al. Demonstration of Krypton: Incremental and Approximate Inference for Faster Occlusion-based Deep CNN Explanations. In *SysML*, 2019.
[20] S. Nakandala et al. Incremental and Approximate Inference for Faster Occlusion-Based Deep CNN Explanations. In *ACM SIGMOD*, 2019.
[21] S. Nakandala et al. Incremental and Approximate Computations for Accelerating Deep CNN Inference. *ACM TODS*, 2020.
[22] S. Nakandala and A. Kumar. Vista: Optimized System for Declarative Feature Transfer from Deep CNNs at Scale. In *ACM SIGMOD*, 2020.
[23] M. Nikolic et al. LINVIEW: Incremental View Maintenance for Complex Analytical Queries. In *ACM SIGMOD*, 2014.
[24] A. Ordookhanians et al. Demonstration of Krypton: Optimized CNN Inference for Occlusion-Based Deep CNN Explanations. In *VLDB*, 2019.
[25] O. Russakovsky et al. Imagenet Large Scale Visual Recognition Challenge. In *International Journal of Computer Vision*, 2015.
[26] T. K. Sellis. Multiple-Query Optimization. In *ACM TODS*, 1988.
[27] P. Voigt and A. Von dem Bussche. *The EU General Data Protection Regulation (GDPR): A Practical Guide.* Springer Publishing Company, Inc., 2017.
[28] Z. Wang et al. Image Quality Assessment: From Error Visibility to Structural Similarity. In *IEEE Transactions on Image Processing*, 2004.
[29] M. D. Zeiler and R. Fergus. Visualizing and Understanding Convolutional Networks. In *European Conference on Computer Vision*, 2014.
[30] C. Zhang et al. Materialization Optimizations for Feature Selection Workloads. In *ACM SIGMOD*, 2014.
[31] L. M. Zintgraf et al. Visualizing Deep Neural Network Decisions: Prediction Difference Analysis. In *ICLR*, 2017.

Technical Perspective: Revealing Every Story of Data in Blockchain Systems

Yaron Kanza
AT&T Labs-Research
kanza@research.att.com

For many applications, data are worthy only if they are trustworthy. The concept of trust is sometimes elusive, and yet it is fundamental in data management. Even when not expressed explicitly, the correctness of computations and reliability of applications depend on trustworthy management of the data. These notions received new attention with the advent of blockchain and distributed ledger technology.

Blockchain was originally introduced as a decentralized ledger of cryptocurrency transactions, in order to solve the "double-spending" problem [2]. Cryptocurrency coins that are given to a user should not be spent more than once. This is crucial for establishing trust in the currency and guaranteeing that the total number of coins will be limited. To prevent double spending, blockchain is tamper-proof and transparent—it is very hard computationally to change stored transactions (practically impossible).

The ability to create a trusted ledger in a decentralized environment, by consensus, attracted the attention of practitioners, theoreticians, organizations and application developers. A large variety of blockchain technologies and blockchain-based applications were developed [3]. But while blockchain technologies have many advantages, they still lack many capabilities that exist in database management systems, e.g., query language, views, data provenance, etc.

The database community has extensively studied data provenance (also known as data lineage) as a concept and a set of tools that are aimed to make data history more transparent [1]. Being able to examine the "story" of a data instance, starting with the data sources and through the operations that were applied to the data, has been promoted as a way to increase credibility and the user's understanding of the data in complex databases.

The paper of Ruan et al. presents a powerful combination of blockchain and provenance. It lays foundations for building a bridge between database systems and blockchains by showing that the marriage of blockchain technologies and database concepts like provenance can yield a better solution for transparent data management, while tracing historical changes in the data.

Originally, blockchains were not designed for tracking historical data. Once an amount of cryptocoins has been spent, it can no longer be a part of a valid payment, so its does not need to be accessed. Furthermore, dependencies between operations are not recorded, e.g., when a value is read from the blockchain, modified and written back to the blockchain, the dependency between the stored values is not recorded. The challenge the authors had to solve was how to provide provenance information in a way that is both trustworthy and efficient. This is not an easy task given that blockchain systems often sacrifice efficiency for reliability and security.

To track provenance data, the authors present novel data structures—a novel index based on skip lists and Merkle DAG which is an adaptation of Merkle tree—and their integration to implement efficient and reliable storage and retrieval of provenance data. To suit the blockchain environment, these index structures are required to satisfy the following three properties. (1) The index should provide a verifiable digest of tracked states, without the need to read the entire transaction history. (2) Updates should be incremental and succinct, because the storage and the management of provenance data must be efficient. (3) The index should be tamper-proof, similar to the transaction and state information for which it was built. The paper shows how to achieve these requirements using the proposed index structures.

The paper demonstrates a clever use of smart contracts to achieve the desired goals. A smart contract is essentially code that is triggered when particular events occur, and executed by the peers that manage the blockchain, in a decentralized fashion. Smart contracts are somewhat similar to a combination of triggers and stored procedures in database management systems, but there are also differences between these mechanisms. Papers like this work of Ruan et al. shed light on some of the differences and similarities between triggers and smart contracts, but more papers like this work are needed to further investigate the limits of smart contracts in data management applications.

This paper is of high significance because it presents a new way to examine how historical data on a blockchain can be retrieved and used—rather than just looking at the latest state, the entire history that led to the state can be examined and used. It paves the way for systems that would manage tamper-proof records of data transformations in a decentralized fashion. This work is also important because it gives us a glimpse into how advanced decentralized data management should look like and how we could increase transparency and trustworthiness in data management.

1. REFERENCES

[1] J. Cheney, L. Chiticariu, and W.-C. Tan. Provenance in databases: Why, how, and where. *Found. Trends Databases*, 1(4):379–474, 2009.

[2] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. Technical report, Manubot, 2008.

[3] S. Underwood. Blockchain beyond bitcoin. *Commun. ACM*, 59(11):15–17, 2016.

Revealing Every Story of Data in Blockchain Systems

Pingcheng Ruan
National University of Singapore
ruanpc@comp.nus.edu.sg

Tien Tuan Anh Dinh
Singapore University of Technology and Design
dinhtta@sutd.edu.sg

Qian Lin
National University of Singapore
linqian@comp.nus.edu.sg

Meihui Zhang
Beijing Institute of Technology
meihui_zhang@bit.edu.cn

Gang Chen
Zhejiang University
cg@zju.edu.cn

Beng Chin Ooi
National University of Singapore
ooibc@comp.nus.edu.sg

ABSTRACT

The success of Bitcoin and other cryptocurrencies bring enormous interest to blockchains. A blockchain system implements a tamper-evident ledger for recording transactions that modify some global states. The system captures the entire evolution history of the states. The management of that history, also known as data provenance or lineage, has been studied extensively in database systems. However, querying data history in existing blockchains can only be done by replaying all transactions. This approach is feasible for large-scale, offline analysis, but is not suitable for online transaction processing.

We present *LineageChain*, a fine-grained, secure, and efficient provenance system for blockchains. *LineageChain* exposes provenance information to smart contracts via simple interfaces, thereby enabling a new class of blockchain applications whose execution logics depend on provenance information at runtime. *LineageChain* captures provenance during contract execution and stores it in a Merkle tree. *LineageChain* provides a novel skip list index that supports efficient provenance queries. We have implemented *LineageChain* on top of Hyperledger Fabric and a blockchain-optimized storage system called ForkBase. We conduct extensive evaluation, demonstrating benefits of *LineageChain*, its efficient querying, and its small storage overhead.

1. INTRODUCTION

Blockchains are capturing attention from both academia and industry. A blockchain is a chain of blocks, in which each block contains many transactions and is linked with the previous block via a hash pointer. It was first used in Bitcoin [12] to store cryptocurrency transactions. Often referred to as decentralized ledger, blockchain ensures integrity (tamper evidence) of the complete transaction history. It is replicated over a peer-to-peer (P2P) network, and a distributed consensus protocol, for instance Proof-of-Work (PoW), is used to ensure that honest nodes in the network have the same ledger. More recent blockchains, for instance Ethereum [1] and Hyperledger Fabric [3], enable applications beyond cryptocurrencies by supporting smart contracts. A smart contract has its states stored on the blockchain. The states are modified via transactions that invoke the contract.

The management of data history, or data provenance, has been extensively studied in databases, and many systems have been designed to support provenance [6, 4]. In the context of blockchain, there is explicit, but only coarse-grained support for data provenance. In particular, the blockchain can be seen as having some states (with known initial values), and every transaction moves the system to new states. The evolution history of the states (or provenance) can be securely and completely reconstructed by replaying all transactions. However, this reconstruction can only be done during offline analysis. During contract execution (or runtime), no provenance information is accessible to smart contracts. This lack of runtime access to provenance therefore restricts the expressiveness of the computation logics that the contract can encode.

Consider an example smart contract shown in Figure 1, which contains a method for transferring a number of tokens from one user to another. Suppose user A wants to send tokens to B based on the latter's historical balance in recent months. For example, A only sends tokens if B's average balance per day is more than t. It is not currently possible to write a contract method for this operation. To work around this, A needs to first compute the historical balance of B by querying and replaying all on-chain transactions, then based on the result issues the `Transfer` transaction. Beside performance overhead incurred from multiple interactions with the blockchain, this approach is not *safe*: it violates transaction serializability. In particular, suppose A issues the `Transfer` transaction tx based on its computation of B's historical balance. But before tx is received by the blockchain, another transaction is committed such that B's average balance becomes $t' < t$. Consequently, when tx is later committed, it will have been based on stale state, and therefore fails to meet the intended business logic. This scenario can be caused by benign network conditions as well as by malicious attacks. In blockchains with native currencies, serializability violation can be exploited for Transaction-Ordering attacks that cause substantial financial loss to the users [10].

We design and implement *LineageChain*, a fine-grained, secure, and efficient provenance system for blockchains that enables a new class of smart contracts which can access provenance information at runtime. *LineageChain* avoids the safety issue of existing solutions, as illustrated in the ex-

©VLDB Endowment 2019. This is a minor revision of the paper entitled "Fine-Grained, Secure and Efficient Data Provenance on Blockchain Systems", published in the Proceedings of the VLDB Endowment, Vol. 12, No. 9, 975-988. DOI: https://doi.org/10.14778/3329772.3329775
.

```
contract Token {
    method Transfer(sender, recipient, amount) {
        bal1 = gState[sender];
        bal2 = gState[recipient];
        if (amount < bal1) {
            gState[sender] = bal1 - amount;
            gState[recipient] = bal2 + amount;
} } }
```

Figure 1: A smart contract for token management.

ample above, by allowing the transaction logic to be based on provenance at runtime. Although our goal is similar to that of existing works in adding provenance to databases, we face three unique challenges due to the nature of blockchain. First, there is a lack of data operators whose semantics capture provenance in the form of input-output dependency. More specifically, for general data management workloads (i.e., non-cryptocurrency), current blockchains expose only generic operators, for example, `put` and `get` of key-value tuples. These operators do not have input-output dependency. In contrast, relational database operators such as `map`, `join`, `union`, are defined as relations between input and output, which clearly capture their dependencies. To overcome this lack of provenance-friendly operators, we instrument the blockchain runtime to record read-write dependency of all the states used in any contract invocation, which is then passed to a user-defined method that specifies which dependency should be persisted.

The second challenge is that blockchains assume an adversarial environment, and therefore any captured provenance must be made tamper evident. To address this, we store provenance in a Merkle tree data structure that also allows for efficient verification. The final challenge is to ensure that provenance queries are efficient, not only to improve latency, but also to avoid degrading security [11]. To address this challenge, we design a novel skip list index optimized for provenance queries.

In summary, we make the following contributions:

- We present *LineageChain*, a system that efficiently captures fine-grained provenance for blockchains. It stores provenance securely, and exposes a simple access interface to smart contracts.
- We present a novel index optimized for querying blockchain provenance. The index is similar to skip list, but is deterministic. Its performance is independent of the blockchain size.
- We implement *LineageChain* for Hyperledger Fabric v1.3 [3]. Our implementation builds on top of ForkBase, a blockchain-optimized storage [14]. Our experimental results demonstrate its benefits to provenance-dependent applications and its efficient querying.

LineageChain is a component of our FabricSharp system [2], for which we improve Fabric's execution and storage layer for the secure runtime provenance support. Elsewhere, we have addressed the consensus bottleneck by applying sharding efficiently and exploiting trusted hardware to scale out system horizontally, to substantially improve the system throughput [7]. We have also improved the storage efficiency by designing a tamper-evident storage engine that supports efficient forking called Forkbase.

2. ORGANIZING BLOCKCHAIN STATES

In this section, we discuss how the global states are organized in blockchains. [8] provides a comprehensive survey of blockchain design. There are three key requirements for building an index over the global states of a blockchain. We explain how they are met in Ethereum and Hyperledger Fabric. *LineageChain* also meets these requirements.

Tamper evidence. A user may read some states without downloading and executing all the transactions. Thus, the index structure must be able to generate an integrity proof for any state. The index must provide a unique digest for the global states, so that blockchain nodes can quickly check if their states are the same.

Incremental update. Whereas the size of global states may be large, one block only updates a small portion of states. For example, some states may be updated at every block, whereas others may be updated much more infrequently. Because the index must be updated at every block, it must be efficient at handling incremental updates.

Snapshot. A snapshot of the index, as well as of the global states, must be made at every block. This is necessary because of the immutability property of the blockchain which allows users to read any historical states. It is also important for block verification: when a new block is received that creates a fork, an old snapshot of the state is used for verification. Even when the blockchain allows no forks, snapshots enable roll-back when the received block is found to be invalid after execution.

Existing blockchains use indices that are based on Merkle trees. In particular, Ethereum implements Merkle Patricia Trie (MPT), and Hyperledger Fabric v0.6 implements Merkle Bucket Tree (MBT). In a Merkle tree, content of the parent node is recursively defined by those of the child nodes. A proof of integrity can be efficiently constructed without reading the entire tree. The Merkle tree meets the first requirement. It also meets the second requirement, because only the tree nodes affected by the update need to be changed. It meets the third requirement, because an update in the block recursively creates new tree nodes in the path affected by the change. And the new root then serves as index of the new snapshot, and is then included in the block header.

3. FINE-GRAINED PROVENANCE

In this section, we describe how we capture provenance, and the smart contract APIs for accessing it. We use a running example of the token smart contract shown in Figure 1. Figure 2 depicts how the global states are modified by the contract. In particular, the contract is deployed at block L in the blockchain. Two accounts (or addresses), *Addr1* and *Addr2*, are initialized with 100 tokens. Two transactions *Txn1* and *Txn2* transferring tokens between the two addresses are committed at blocks M and N respectively. The value of *Addr1* is 100 from block L to block $M-1$, 90 from block M to $N-1$, and 70 from block N. The global state *gState* is essentially a map of addresses to their values.

3.1 Capturing Provenance

In *LineageChain*, every contract method can be made provenance-friendly via a *helper* method. In particular, during transaction execution, *LineageChain* collects the identi-

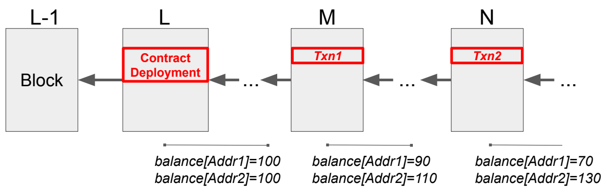


Figure 2: Content of the blockchain and *gState*.

```
contract Token {
    method Transfer(...){...} // as above
    method prov_helper(name, reads, writes) {
      if name == "Transfer" {
        for (id,value) in writes {
            if (reads[id] < value) {
                recipient = id;
            } else {sender = id; }
        }
        // dependency list with a
        // single element.
        dep = [sender];
        return {recipient:dep};
      }
      ...
    }
}
```

Figure 3: The provenance helper method for *Token* contract, which defines dependency between the sender identifier and recipient identifier.

fiers and values of the accessed states, i.e., those used in `read` and `write` operations. The results are a read set `reads` and write set `writes`. For *Txn1*, `reads` $= \{Addr1 : 100, Addr2 : 100\}$, and `writes` $= \{Addr1 : 90, Addr2 : 110\}$. After the execution finishes, these sets are passed to `prov_helper` method, together with the name of the contract method. `prov_helper` has the following signature:

```
method prov_helper(name: string,
                   reads: map(string, byte[]),
                   writes: map(string, byte[]))
      returns map(string, string[]);
```

`prov_helper` is defined by the contract developer, and it returns a set of dependencies based on the input read and write sets. Figure 3 shows an implementation of the helper method for the Token contract. It first computes the identifier of the sender and recipient from the read and write sets. Specifically, the identifier whose value in `writes` is lower than that in `reads` is the sender, and the opposite is true for the recipient. It then returns a dependency set of a single element: the recipient-sender dependency. In our example, for *Txn1*, this method returns $\{Addr2 : [Addr1]\}$.

3.2 Smart Contract APIs

Current smart contracts can only access the *latest* states. *LineageChain* provides access to the captured provenance via three additional smart contract APIs.

- `Hist(stateID, [blockNum])`: returns the tuple `(val, blkStart, txnID)` where `val` is the value of `stateID` at block `blockNum`. If `blockNum` is not specified, the latest block is used. `txnID` is the transaction that sets `stateID` to `val`, and `blkStart` is the block number at which `txnID` is executed.
- `Backward(stateID, blkNum)`: returns a list of tuples `(depStateID, depBlkNum)` where `depStateID` is the dependency state of `stateID` at block `blkNum`. `depBlkNum` is the block number at which the value of `depStateID` is set. In our example, `Backward(Addr2, N)` returns `(Addr1, M)`.
- `Forward(stateID, blkNum)`: similar to the `Backward` API, but returns the states of which `stateID` is a dependency. For example, `Forward(Addr1, L)` returnss `(Addr2, M)`.

```
contract Token {
  ...
  method Blacklist(addr) {
    blk := last block in the ledger
    blacklisted = false;
    iterate 5 times {
      val, startBlk, txnID = Hist(addr, blk);
      for (depAddr, depBlk)
            in (Backward(addr, startBlk)
             or Forward(addr, startBlk)) {
        if depAddr in gState["blacklist"] {
          gState["blacklist"].append(addr);
          return;
        }
      }
      blk = startBlk - 1;
    }
  }
}
```

Figure 4: Smart contract with the new APIs.

Figure 4 illustrates how the above APIs are used to express smart contract logics that are currently impossible, as shown in the `Blacklist` method. This will mark an address as blacklisted if one of its last 5 transactions is with a blacklisted address.

4. PROVENANCE STORAGE AND QUERY

In this section, we describe the design for storing and querying the captured provenance.

4.1 Storage

LineageChain enhances existing blockchain storage layer to provide efficient tracking and tamper evidence for the captured provenance. The key idea is to reorganize the flat leaf nodes in the original Merkle tree into a Merkle DAG.

Merkle DAG. Let k be the unique identifier of a blockchain state, whose evolution history is expected to be tracked. Let v be the unique version number that identifies the state in its evolution history. When the state at version v is updated, the new version v' is strictly greater than v. In *LineageChain*, we directly use the block number as its version v. Let $s_{k,v}$ denote the value of the state with identifier k at version v. We drop the subscripts if the meaning of k and v are trivial. For any $k \neq k'$ and $v \neq v'$, $s_{k,v}$ and $s_{k',v'}$ represent the values of two different states at different versions. s_k^b represents the state value with identifier k at its latest version before block b. In our example, for $k = Addr1$ and $v = M$, $s_{k,v} = 90$.

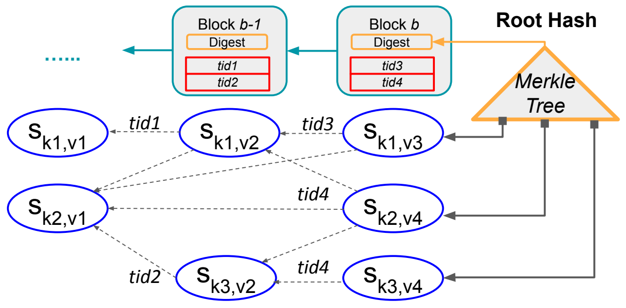


Figure 5: A Merkle DAG for storing provenance. s_{k_2,v_4} and s_{k_3,v_4} updated by the same transaction (*tid*$_4$), block b contains two transactions, *tid*$_3$ and *tid*$_4$. Its latest states are represented by the Merkle root.

Definition 1. *A transaction, identified by tid which is strictly increasing, consumes a set of input states S^i_{tid} and produces a set of output states S^o_{tid}. A valid transaction satisfies the following properties:*

$$\forall s_{k_1,v_1}, s_{k_2,v_2} \in S^o_{tid}.\quad k_1 \neq k_2 \wedge v_1 = v_2 \tag{1}$$

$$\forall s_{k_1,v_1} \in S^i_{tid}, s_{k_2,v_2} \in S^o_{tid}.\quad v_1 < v_2 \tag{2}$$

$$\forall s_{k,v} \in S^i_{tid}, s_{k,v'} \in S^i_{tid'}.\quad tid < tid' \Rightarrow v \leq v'. \tag{3}$$

$$tid \neq tid' \Rightarrow S^o_{tid} \cap S^o_{tid'} = \emptyset \tag{4}$$

Property (1) means that the versions of all output states of a transaction are identical, because they are updated by the same transaction in the same block. Property (2) implies the version of any input state is strictly lower than that of the output version. This makes sense because the blockchain establishes a total order over the transactions, and because the input states can only be updated in previous transactions. Property (3) specifies that, for all the states with the same identifier, the input of later transactions can never have an earlier version. This ensures the input state of any transaction must be up-to-date during execution. Finally, Property (4) means that every state update is unique.

Definition 2. *The dependency of state s is a subset of the input states of the transaction that outputs s. More specifically:*

$$dep(s) \subset S^i_{tid} \text{ where } s \in S^o_{tid}.$$

Note that *dep*, which is returned by `prov_helper` method, is only a subset of the read set.

Definition 3. *The entry $E_{s_{k,v}}$ of the state $s_{k,v}$ is a tuple containing the current version, the state value, and the hashes of the entries of its dependent state. More specifically:*

$$E_{s_{k,v}} = \langle v, s_{k,v}, \{hash(E_{s'}) | s' \in dep(s_{k,v})\}\rangle$$

An entry uniquely identifies a state. In *LineageChain*, we associate each entry with its corresponding hash.

Definition 4. *The set of latest states at block b, denoted as $S_{latest,b}$, is:*

$$S_{latest,b} = \bigcup_k \{s^b_k\}$$

Let U_b be the updated states in block b. We can compute $S_{latest,b}$ by recursively combining U_b with $S_{latest,b-1} \setminus U_b$.

Definition 5. *χ_b is the root of a Merkle tree built on the map S_b where*

$$S_b = \{k : hash(E_{s^b_k}) | \forall s^b_k \in S_{latest,b}\}.$$

LineageChain stores χ_b as the state digest in the block header.

Forward tracking. One problem with the above DAG model is that it does not support forward tracking, because the hash pointers only reference *backward* dependencies. When a state is updated, these backward dependencies are permanently established, so that they belong to the immutable history of the state. However, the state can be read by future transactions, and as a consequence its forward dependencies cannot be determined at the time of update.

Fortunately, an important observation is that only forward dependencies of the *latest state* are mutable. Once the state is updated, due to the execution model of blockchain smart contract, in which the latest state is always read, forward dependencies of the previous state version become permanent. As a result, they can be included into the history. Figure 5 illustrates an example, in which forward dependencies of s_{k_1,v_1} become fixed when the state is updated to s_{k_1,v_2}. This is because when the transaction that outputs s_{k_2,v_4} is executed, it reads s_{k_1,v_2} instead of s_{k_1,v_1}.

In *LineageChain*, for each state $s_{k,v}$ at its latest version, we buffer a list of forward pointers to the entries whose dependencies include $s_{k,v}$. We refer to this list as $F_{s_{k,v}}$, where

$$F_{s_{k,v}} = \{hash(E_{s'}) | s_{k,v} \in dep(s')\}$$

When the state is updated to $s_{k,v'}$ for $v' > v$, we store $F_{s_{k,v}}$ at the entry of $s_{k,v'}$.

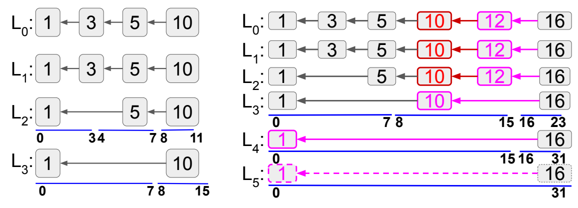
4.2 Efficient Query Processing

The Merkle DAG structure supports efficient access to the latest state version, since the state index at block b contains pointers to all the latest versions at this block. To read the latest version of s, one simply reads χ_b, follows the index to the entry for s, and then reads the state value from the entry. However, querying an arbitrary version in the DAG is inefficient, because one has to start at the DAG root and traverse the edges towards the requested version. Supporting fast version queries is important when a user wants to examine the state history only from a specific version (for auditing, for example). It is also important for provenance-dependent smart contracts because such queries directly affect contract execution time.

Deterministic Append-only Skip List. We propose to build an index, called Deterministic Append-only Skip List (or DASL), on top of a state DAG to support fast version queries. The index has a skip list structure. It is designed for blockchains, exploiting the fact that the blockchain is append-only, and randomness is not well supported [5]. A DASL has two distinct properties compared to a normal skip list. First, it is append-only; that is, the index keys of the

```
struct Node {
  Version v;
  Value val;
  List<Version> pre_versions;
  List<Node*> pre_nodes;
}
```

Figure 6: A Node structure that captures a state $s_{k,v}$ with value *val*



(a) (b)

Figure 7: (a) A DASL containing versions 1, 3, 5 and 10. The base b is 2. The intervals for L_2 and L_3 are shown in blue lines. (b) The new DASL after appending versions 12 and 16. L_4 is created when appending version 16. L_5 is created, then discarded.

appended entries, which are state versions, are strictly increasing. Second, it is deterministic; that is, the index structure is uniquely determined by the values of the appended items, unlike a stochastic skip list.

Definition 6. *Let $V_k = \langle v_0, v_1, ... \rangle$ be the sequence of version numbers of states with identifier k, in which $v_i < v_j$ for all $i < j$. A DASL index for k consists of N linked lists $L_0, L_1, .., L_{N-1}$. Let v^i_{j-1} and v^i_j be the versions in the $(j-1)^{th}$ and j^{th} node of list L_i. Let b be the base number, a system-wide parameter. The content of L_i is constructed as follows:*

1) $v_0 \in L_i$

2) Given v^i_{j-1}, v^i_j is the smallest version in V_k such that:

$$\left\lfloor \frac{v^i_{j-1}}{b^i} \right\rfloor < \left\lfloor \frac{v^i_j}{b^i} \right\rfloor \qquad (5)$$

Figure 6 shows how DASL is stored with the state in a data structure called Node. This structure (also referred to as node) contains the state version and value. A node belongs to multiple lists (or levels), hence it maintains a list of pointers to other nodes in each level as well as a list of the version numbers of pointed nodes. Both lists are of size N, and the i^{th} entry of a list points to the previous version (or the previous node) of this node in level L_i. For the same key, the version number uniquely identifies the node, and hence we use version numbers to refer to the corresponding nodes.

We can view a list as consisting of continuous, non-overlapping intervals of certain sizes. In particular, the j^{th} interval of L_i represents the range $R^i_j = [jb^i, (j+1)b^i)$. Only the smallest version in V_k that falls in this range is included in the list. Figure 7(a) gives an example of a DASL structure with $b = 2$. It can be seen that when the version numbers are sparsely distributed, the lists at lower levels are identical. In this case, b can be increased to create larger intervals in order to reduce the overlapping among lower-level lists.

Algorithm 1: DASL Append

```
   Input: version $v$ and last node $last$
   Output: previous versions and nodes
1  $level$=0; // list level
2  $pre\_versions$ = [];
3  $pre\_nodes$ = [];
4  $finish$ = false ;
5  $cur$ = $last$ ;
6  while not $finish$ do
7      $l$ = $cur$->$pre\_versions.size()$ ;
8      if $l > 0$ then
9          for $j$=$level$; $j$<$l$; ++$j$ do
10             if $cur$->$version$ / $b^j$ < $v$ / $b^j$ then
11                 $pre\_versions$.append($cur$->$version$);
12                 $pre\_nodes$.append($cur$);
13             else
14                 $finish$ = true;
15                 break;
16         if not $finish$ then
17             $cur$ = $cur$->$pre\_versions[l-1]$;
18             $level$ = $l$
19     else
           /* We have reached the last level        */
20         $finish$ = true;
21         while $cur$->$version$ / $b^{level}$ < $v$ / $b^{level}$ do
22             ++$level$;
23             $pre\_versions$.append($cur$->$version$);
24             $pre\_nodes$.append($cur$);
25 return $pre\_version$, $pre\_nodes$;
```

A DASL and a skip list share two properties. First, if a version number appears in L_i, it also appears in L_j where $j < i$. Second, with $b = 2$, suppose the last level that a version appears in is i, then this version's preceding neighbour in L_i appears in L_j where $j > i$. Given these properties, a query for a version in the DASL is executed in the same way as in the skip list. More specifically, the query traverses a high-level list as much as possible, starting from the last version in the last list. It moves to a lower level only if the preceding version in the current list is strictly smaller than the requested version. In DASL, the query for version v_q returns the largest version $v \in V_k$ such that $v \leq v_q$ (the inequality occurs when v_q does not exist). This result represents the value of the state which is visible at the time of v_q.

We now describe how a new node is appended to DASL. The challenge is to determine the lists that should include the new node. Algorithm 1 details the steps that find the lists, and subsequently the previous versions, of the new node. The key idea is to start from the last node in L_0, and then keep moving up the list level until the current node and the new node belong to the same interval (line 9 - 18). Figure 7(b) shows the result of appending a node with version 12 to the original DASL. The algorithm starts at node 10 and moves up to lists L_1 and L_2. It stops at L_3 because in this level nodes 10 and 12 belong to the same interval, i.e., [8, 16). Thus, the new node is appended to lists L_0 to L_2. When the algorithm reaches the last level and is still able to append, it creates a new level where node 0 is the first entry and repeats the process (line 21 - 24). In Figure 7(b), when appending version 16, all existing lists can be used. The algorithm then creates L_4 with node 1 and appends node 16 to it. It also creates a new level L_5, but subsequently discards it because node 16 will not be appended since it belongs to same interval of $[0, 32)$ with node 1.

4.3 Discussion

Our new Merkle DAG can be easily integrated to existing blockchain index structures. It meets the three requirements listed in Section 2. In particular, existing Merkle indices such as MPT store state values directly at the leaves, whereas the Merkle DAG in *LineageChain* stores the entry hashes of the latest state versions at the leaves. By adding one more level of indirection, we preserve the three properties of the index (tamper evidence, incremental update and snapshot), while enhancing it with the ability to traverse the DAG to extract fine-grained provenance information. Recall that the state entry hash captures the entire evolution history of the state. Since this hash is protected by the Merkle index for tamper evidence, so is the state history. In other words, we add integrity protection for provenance without any extra cost to the index structure. For example, suppose a client wants to read a specific version of a state, it first reads the state entry hash at the latest block. This read operation can be verified against tampering, as in existing blockchains. Next, the client traverses the DAG from this hash to read the required version. Because the DAG is tamper evident, the integrity of the read version is guaranteed.

DASL and Merkle DAG integration. Adding DASL to the Merkle DAG is straightforward. The node structure (Figure 6) is stored in the state entry (Definition 3). The node pointers are implemented as entry hashes. The Merkle tree structure remains unchanged.

Speed vs. storage. As a skip list variant, DASL shares the same space complexity and logarithmic query time complexity. Suppose there are v^* number of versions and the base of DASL is b. There are at most $\lceil \log_b v^* \rceil$ levels and the i-th level takes at most $\lceil \frac{v^*}{b^i} \rceil - 1$ pointers. Suppose the queried version is v^q and the query distance $d = v^* - v^q$, the maximum number of hops in such query is capped at $2b\lceil \log_b d \rceil$. This is because a typical query consists of two stages: one going towards the lower levels, and the other going towards the upper level. Each stage involves traversing at most b hops on the same list before moving to the next level, and there are at most $\lceil \log_b d \rceil$ levels. It can be seen that b determines the tradeoff between the space overhead and query latency. Furthermore, DASL queries are more efficient for more recent versions, i.e. d are small, which is useful for smart contracts that rely on recent rather than old versions. Finally, the performance of such recent-version queries does not change as the state history grows.

5. PERFORMANCE EVALUATION

We implement *LineageChain* on top of Hyperledger Fabric v1.3. More details of the implementation can be found in [13]. Figure 8 shows the software stack, highlighting the changes to the original Fabric's stack. We completely replace Fabric's storage layer with our implementation of the Merkle DAG and DASL index on top of ForkBase [14], a state-of-the-art blockchain storage system with efficient support for versioning. We instrument Fabric's execution engine

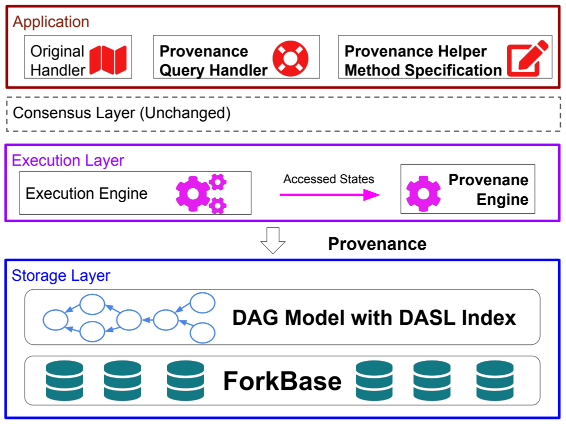


Figure 8: *LineageChain*'s software stack. The original storage layer is replaced with the implementation that supports fine-grained provenance. The original execution layer is instrumented with a provenance capture engine. The application layer contains the new helper method and provenance query APIs. The consensus layer is unchanged.

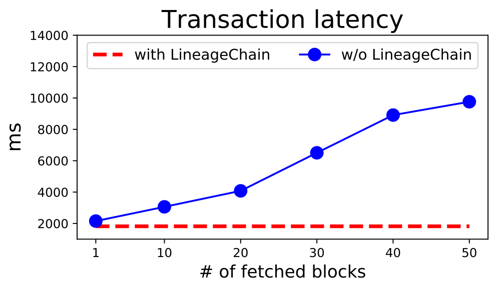
to record read and write sets during contract execution. At the application layer we add a new helper method and three provenance APIs. The execution engine invokes the helper method after every successful contract execution.

5.1 Methodologies

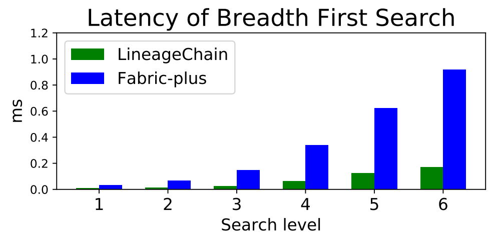
We evaluate *LineageChain* against two baselines. In the first baseline, called *Fabric-plus*, we directly store provenance information to Fabric's original storage and rely on its internal index to support provenance query. In the second baseline, called *LineageChain-lite*, we use ForkBase for storing state versions. This baseline has no support for multi-state dependency, and no DASL index. We use this to understand the index's performance.

We perform three sets of experiments. First, we evaluate the performance of *LineageChain* for provenance-dependent blockchain applications. We compare it against the approach that queries provenance offline before issuing blockchain transactions. Second, we evaluate the performance of provenance queries in *LineageChain* on a single machine. For single-state version queries, we use the YCSB benchmark provided in BLOCKBENCH [9] to populate the blockchain states with key-value tuples. We then measure the latencies of two queries: one retrieves a state at a specific block, and the other iterates over the state history. For multi-state dependency tracking, we implement a contract for a supply chain application. In this application, a phone is assembled from intermediary components which are made from other components or raw material. The supply chain creates a DAG representing the derivation history of a phone. The maximum depth of the DAG is 6. We generate synthetic data for this contract, and examine the latency of the operation that uses `Backtrack` to retrieve dependencies of a given phone.

In the third set of experiments, we evaluate the impact of provenance on the overall blockchain performance. For this, we run the Smallbank benchmark on multiple nodes.



(a)



(b)

Figure 9: Performance of (a) a provenance-dependent blockchain application and (b) BFS Traversal latency

We measure the overall throughput, and analyze the cost breakdown to understand the overhead of provenance support.

Our experiments are run on a local cluster of 16 nodes. Each node is equipped with E5-1650 3.5GHz CPU, 32GB RAM, and 2TB hard disk. The nodes are connected via 1Gbps Ethernet.

5.2 Experimental Results

Provenance-dependent Applications

We implement a simple provenance-dependent blockchain application by modifying the YCSB benchmark in BLOCKBENCH such that the update operation depends on historical values. With *LineageChain*, the contract has direct access to the provenance information, and the client remains the same as in the original YCSB. Without *LineageChain*, the client is modified such that it reads B latest blocks before issuing transactions. B represents how far behind the client is to the latest states.

Figure 9(a) shows transaction latency with varying B. It can be seen that with *LineageChain*, the latency remains almost constant because the client does not have to fetch any block for the provenance query. In contrast, without *LineageChain*, the latency increases linearly with B. This demonstrates the performance gain brought by *LineageChain* for having access to provenance information at runtime.

Provenance Queries

We first create 500 key-value tuples and then continuously issue update transactions until there are more than 10k blocks in the ledger. Each block contains 500 transactions. We then execute a query for the values of a key at different block numbers. Figure 10(a) illustrates the query latency with increasing block distance from the last block. It can be seen that when the distance is small, *LineageChain-lite* has the lowest latency. *LineageChain-lite* does not have DASL index, and as a consequence for this query it has to scan linearly from the latest version. As expected, the query is fast when the requested version is very recent because the number of reads is small, but degrades the performance quickly as the distance increases. In particular, when the block distance reaches 128, the query is $4\times$ slower than *LineageChain*. We observe that the query latency in *Fabric-plus* is independent of the block distance, because the query uses flat storage index directly. *LineageChain* outperforms both *LineageChain-lite* and *Fabric-plus*. Because of DASL, the query latency in *LineageChain* is low when the block distance is small. When the block distance increases, the latency increases only logarithmically, as opposed to linearly in *LineageChain-lite*.

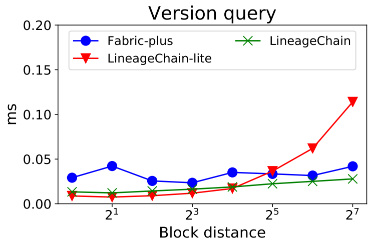
We repeat the experiment above while fixing the block distance to 64 and varying the total number of blocks. Figure 10(b) shows the results for the version query with increasing number of blocks. It can be seen that the query latency in both *LineageChain* and *Fabric-plus* remains roughly the same. In other words, the performance of version queries in these systems are independent of the block numbers, which is due to the DAG data model that tracks state versions. *LineageChain* outperforms *Fabric-plus*, thanks to the index that reduces the number of entries needed to be read.

Next, we measure the latency for the operation that scans the entire version history of a given key. Figure 10(c) shows the scan latency with increasing number of blocks. For *Fabric-plus*, we first construct the key range and rely on the storage iterator for scanning. *LineageChain-lite* and *LineageChain* both use ForkBase iterator, and therefore they have the same performance. As the number of blocks increases, the version history becomes longer which accounts for the linear increase in latency in both systems. However, *LineageChain* outperforms *Fabric-plus* by a constant factor. We attribute this difference to ForkBase's optimizations for version tracking.

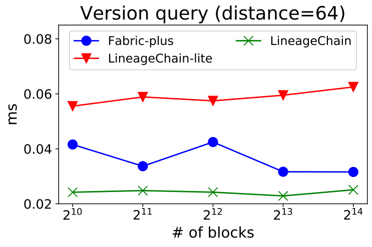
Finally, we evaluate the query performance with multi-state dependency. We populate the blockchain states and issue transactions that produce new phones. We perform a breadth-first search to retrieve all the dependencies of a phone. For this experiment, we only compare *Fabric-plus* and *LineageChain*, because *LineageChain-lite* does not support multi-state dependencies. Figure 9(b) shows the performance with varying search depths. The latency of both *Fabric-plus* and *LineageChain* grow exponentially with increasing depths, but *LineageChain* outperforms the baseline. It is because the index in *LineageChain* directly captures the dependencies, whereas each backtrack operation in *Fabric-plus* requires traversing the storage index. As the number of queries increases with the search level, their performance gap widens.

Performance Overhead

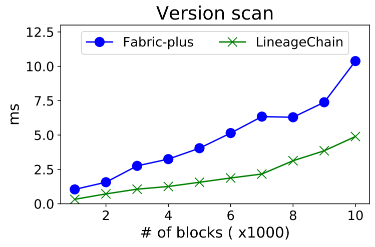
Finally, we evaluate *LineageChain* overhead on Hyperledger Fabric v1.3. We use 16 nodes and vary the offer load by increasing the client's transaction rate. Figure 11 shows the performance overhead. At saturation, *LineageChain-lite* and *LineageChain* add less than $200ms$ in latency, compared to the original Fabric that has no provenance support. In contrast, *Fabric-plus* adds more than $1s$. *LineageChain-lite*



(a)



(b)



(c)

Figure 10: Latency of the version query on YCSB with increasing block distance (a) and increasing number of blocks (b). Latency of the version scan with increasing block number (c).

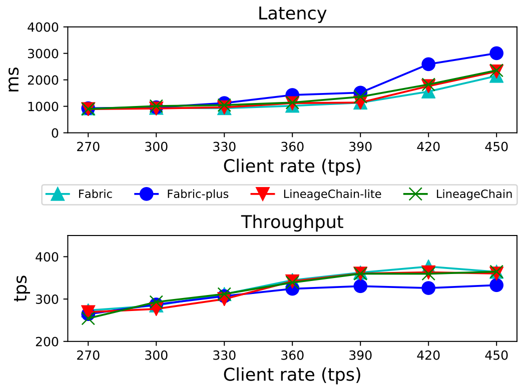


Figure 11: Performance on Fabric v1.3

and *LineageChain* reach similar throughput as the original Hyperledger, which is around 350tps. *Fabric-plus* peaks at around 330tps. These results demonstrate that *LineageChain*'s overhead over the original Fabric is small.

6. CONCLUSIONS

In this paper, we presented *LineageChain*, a fine-grained, secure and efficient provenance system for blockchains. The system efficiently captures provenance information during runtime, and exposes simple APIs to smart contracts, which enables provenance-dependent blockchain applications. Provenance is stored securely, and queries are efficient thanks to a novel skip list index. We implemented *LineageChain* on top of Hyperledger Fabric and benchmarked it against several baselines. The results show the benefits of *LineageChain* in supporting rich, provenance-dependent applications. They demonstrate that provenance queries are efficient, and *LineageChain* incurs small runtime overhead.

7. ACKNOWLEDGMENTS

This research is supported by Singapore Ministry of Education Academic Research Fund Tier 3 under MOE's official grant number MOE2017-T3-1-007.

8. REFERENCES

[1] Ethereum. `https://www.ethereum.org`.

[2] FabricSharp. `https://www.comp.nus.edu.sg/~dbsystem/fabricsharp/`.

[3] Hyperledger fabric. `https://github.com/hyperledger/fabric`.

[4] P. Buneman, A. Chapman, and J. Cheney. Provenance management in curated databases. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, pages 539–550. ACM, 2006.

[5] C. Cachin, S. Schubert, and M. Vukolić. Non-determinism in byzantine fault-tolerant replication. *arXiv preprint arXiv:1603.07351*, 2016.

[6] L. Chiticariu, W.-C. Tan, and G. Vijayvargiya. Dbnotes: a post-it system for relational databases based on provenance. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, pages 942–944. ACM, 2005.

[7] H. Dang, T. T. A. Dinh, D. Loghin, E.-C. Chang, Q. Lin, and B. C. Ooi. Towards scaling blockchain systems via sharding. *Proceedings of the 2019 ACM SIGMOD International Conference on Management of Data*, pages 123–140, 2019.

[8] T. T. A. Dinh, R. Liu, M. Zhang, G. Chen, B. C. Ooi, and J. Wang. Untangling blockchain: A data processing view of blockchain systems. *IEEE Transactions on Knowledge and Data Engineering*, 30(7):1366–1385, 2018.

[9] T. T. A. Dinh, J. Wang, G. Chen, R. Liu, B. C. Ooi, and K.-L. Tan. Blockbench: A framework for analyzing private blockchains. In *Proceedings of the 2017 ACM SIGMOD International Conference on Management of Data*, pages 1085–1100. ACM, 2017.

[10] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 254–269. ACM, 2016.

[11] L. Luu, J. Teutsch, R. Kulkarni, and P. Saxena. Demystifying incentives in the consensus computer. In *Proceedings of the 2015 ACM SIGSAC Conference on Computer and Communications Security*, pages 706–719, 2015.

[12] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. `https://bitcoin.org/bitcoin.pdf`, 2009.

[13] P. Ruan, G. Chen, T. T. A. Dinh, Q. Lin, B. C. Ooi, and M. Zhang. Fine-grained, secure and efficient data provenance on blockchain systems. *PVLDB*, 12(9):975–988, 2019.

[14] S. Wang, T. T. A. Dinh, Q. Lin, Z. Xie, M. Zhang, Q. Cai, G. Chen, B. C. Ooi, and P. Ruan. Forkbase: An efficient storage engine for blockchain and forkable applications. *PVLDB*, 11(10):1137–1150, 2018.