SIGMOD Officers, Committees, and Awardees

Chair	Vice-Chair	Secretary/Treasurer
Donald Kossmann Systems Group ETH Zürich Cab F 73 8092 Zuerich SWITZERLAND +41 44 632 29 40 <donaldk AT inf.ethz.ch>	Anastasia Ailamaki School of Computer and Communication Sciences, EPFL EPFL/IC/IIF/DIAS Station 14, CH-1015 Lausanne SWITZERLAND +41 21 693 75 64 <natassa AT epfl.ch>	Magdalena Balazinska Computer Science & Engineering University of Washington Box 352350 Seattle, WA USA +1 206-616-1069 <magda AT cs.washington.edu>

SIGMOD Executive Committee:
Donald Kossmann (Chair), Anastasia Ailamaki (Vice-Chair), Magdalena Balazinska, K. Selçuk Candan, Yanlei Diao, Curtis Dyreson, Yannis Ioannidis, Christian Jensen, and Jan Van den Bussche.

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

SIGMOD Information Director:
Curtis Dyreson, Utah State University <curtis.dyreson AT usu.edu>

Associate Information Directors:
Huiping Cao, Manfred Jeusfeld, Asterios Katsifodimos, Georgia Koutrika, Wim Martens

SIGMOD Record Editor-in-Chief:
Yanlei Diao, University of Massachusetts Amherst <yanlei AT cs.umass.edu>

SIGMOD Record Associate Editors:
Vanessa Braganholo, Marco Brambilla, Chee Yong Chan, Rada Chirkova, Zachary Ives, Anastasios Kementsietsidis, Jeffrey Naughton, Frank Neven, Olga Papaemmanouil, Aditya Parameswaran, Alkis Simitsis, Wang-Chiew Tan, Nesime Tatbul, Marianne Winslett, and Jun Yang

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

PODS Executive Committee:
Jan Van den Bussche (Chair), Tova Milo, Diego Calvanse, Wang-Chiew Tan, Rick Hull, Floris Geerts

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

Awards Committee:
Surajit Chaudhuri (Chair), David Dewitt, Martin Kersten, Maurizio Lenzerini, Jennifer Widom

Jim Gray Doctoral Dissertation Award Committee:
Ashraf Aboulnaga (co-Chair), Juliana Freire (co-Chair), Kian-Lee Tan, Andy Pavlo, Aditya Parameswaran, Ioana Manolescu, Lucian Popa, Chris Jermaine, Renée Miller

SIGMOD Systems Award Committee:
Mike Stonebraker (Chair), Make Cafarella, Mike Carey, Yanlei Diao, Paul Larson

SIGMOD Officers, Committees, and Awardees (continued)

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. Formerly known as the "SIGMOD Innovations Award", it now honors Dr. E. F. (Ted) Codd (1923 - 2003) who invented the relational data model and was responsible for the significant development of the database field as a scientific discipline. Recipients of the award are the following:

Michael Stonebraker (1992)
David DeWitt (1995)
Serge Abiteboul (1998)
Rudolf Bayer (2001)
Ronald Fagin (2004)
Jennifer Widom (2007)
Umeshwar Dayal (2010)
Stefano Ceri (2013)
Gerhard Weikum (2016)

Jim Gray (1993)
C. Mohan (1996)
Hector Garcia-Molina (1999)
Patricia Selinger (2002)
Michael Carey (2005)
Moshe Y. Vardi (2008)
Surajit Chaudhuri (2011)
Martin Kersten (2014)

Philip Bernstein (1994)
David Maier (1997)
Rakesh Agrawal (2000)
Don Chamberlin (2003)
Jeffrey D. Ullman (2006)
Masaru Kitsuregawa (2009)
Bruce Lindsay (2012)
Laura Haas (2015)

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)

SIGMOD Contributions Award

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

Maria Zemankova (1992)
Jeffrey Ullman (1996)
Raghu Ramakrishnan (1999)
Daniel Rosenkrantz (2001)
Surajit Chaudhuri (2004)
Hans-Jörg Schek (2007)
David Lomet (2010)
H.V. Jagadish (2013)
Samuel Madden (2016)

Gio Wiederhold (1995)
Avi Silberschatz (1997)
Michael Carey (2000)
Richard Snodgrass (2002)
Hongjun Lu (2005)
Klaus R. Dittrich (2008)
Gerhard Weikum (2011)
Kyu-Young Whang (2014)

Yahiko Kambayashi (1995)
Won Kim (1998)
Laura Haas (2000)
Michael Ley (2003)
Tamer Özsu (2006)
Beng Chin Ooi (2009)
Marianne Winslett (2012)
Curtis Dyreson (2015)

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

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

[Last updated : March 31, 2017]

Editor's Notes

Welcome to the March 2017 issue of the ACM SIGMOD Record!

The new year of 2017 begins with a special issue on the **2016 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 potentially to the broader ACM community.

The award committee and editorial board included Zack Ives, Jeff Naughton, Wang-Chiew Tan, and Yanlei Diao. We solicited articles from PODS 2016, SIGMOD 2016, VLDB 2016, ICDE 2016, EDBT 2016, and ICDT 2016, as well as from community nominations. Through a detailed review process five articles were finally selected as 2016 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 our associate editor or an external expert on a given research topic. 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 2016 research highlights cover a broad set of topics, including (a) a new theoretical framework that unifies multiple computation problems in Computer Science and solves them using database technology ("Juggling Functions Inside a Database); (b) a theoretical milestone that closes the gap in our understanding of minimizing tree patterns in querying graph and tree structured data ("Optimizing Tree Patters for Querying Graph- and Tree-Structured Data"); (c) a new query engine that provides efficient, scalable support of aggregate constraints on query answers ("A Scalable Execution Engineer for Package Queries); (d) a new efficient approach to online aggregation by using indexes and making a random walk in the data join graph ("Wander Join and XDB: Online Aggregation via Random Walks); (e) a new implementation that expedites linear algebra operations prevalent in machine learning by adapting database techniques such as column-based compression and sampling-based cost estimation ("Scaling Machine learning via Compressed Linear Algebra").

Finally, this special issue closes with a message from the Editor-in-Chief of ACM TODS.

On behalf of the SIGMOD Record Editorial Board, I hope that you enjoy reading the March 2017 issue of the SIGMOD Record!

Your submissions to the SIGMOD Record are welcome via the submission site:
http://sigmod.hosting.acm.org/record

Prior to submission, please read the Editorial Policy on the website of the SIGMOD Record:
http://sigmod.org/sigmodrecord/authors/

Yanlei Diao

March 2017

Past SIGMOD Record Editors:

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

Technical Perspective: Juggling Functions Inside a Database

Dan Olteanu
University of Oxford
dan.olteanu@cs.ox.ac.uk

The paper entitled "Juggling Functions Inside a Database" gives a brief overview of FAQ, a framework for computational problems expressed as **F**unctional **A**ggregate **Q**ueries. This work falls into my bucket of select database research contributions that go significantly beyond the state of the art along several dimensions. First, it provides an elegant and declarative formalism for a host of ubiquituous computational problems across Computer Science and at the right level of abstraction that exposes structural properties of the problem instances and allows for fine-grained complexity analysis. Second, it is technically deep, proposing an algorithmic solution that achieves lower than or the same complexity as specialized approaches in their respective domain. Third, it is implemented in a commercial database system with scores of real-world applications. Fourth, it is currently applied to in-database analytics and I expect more applications will manifest themselves in the near future.

By unifying many problems under the same formalism, FAQ bears the promise of accelerating research: Scalable data management solutions developed by our community for aggregates over joins, e.g., incremental view maintenance, index data structures, or distributed processing, may become general-purpose solutions for problems outside databases.

I will next expand on some of its contributions.

Unified approach to a host of computational problems. FAQ captures problems in relational databases, logic, matrix and tensor computation, probabilistic graphical models, constraint satisfaction, and signal processing. For instance, FAQ expressions represent queries with joins and aggregates in relational databases, matrix chain computation, maximum a posteriori and marginal distribution queries in probabilistic graphical models.

Technical contribution beyond state of the art. FAQ exploits recent groundbreaking developments on worst-case optimal join algorithms, started by researchers including the FAQ authors, and asymptotically tight bounds on join processing time and result size. This leads to lower complexities for long-standing problems, including counting quantified conjunctive queries in logic and inference in probabilistic graphical models.

FAQ also recovers database techniques, such as pushing aggregates past joins and computing aggregates over factorized joins.

Commercial deployment. FAQ can be easily plugged into existing relational database systems as a standalone library, since it mainly performs query rewriting; however, to attain its low complexity for queries with cycles, an optimal multi-way join algorithm would be also needed. It is in fact already deployed in the LogicBlox engine. FAQ expressions are translated into optimized programs consisting of strata of Datalog-like rules expressing joins over extensional and intensional predicates closed by aggregates with free variables. Although the paper does not report on performance of the FAQ implementation within LogicBlox, it is conceivable that its relative performance over existing solutions follows the reported complexity gap.

Bright future ahead. In-database analytics are a new application of FAQ, where optimization problems are pushed inside the database. The motivation for this application is twofold. First, since data usually resides inside the database, bringing the analytics closer to the data saves export/import time at the interface between database systems and statistical packages. Second, large chunks of machine learning code can be phrased as FAQ expressions!

In conclusion, this paper reports on a well-rounded work of both theoretical and practical relevance. It represents a significant improvement over the state of the art. While a database problem at its core, it can effectively accelerate research across Computer Science. It is an excellent lesson of elegance and technical mastery and I hope you will enjoy learning from it as much as I did.

Juggling Functions Inside a Database

Mahmoud Abo Khamis
LogicBlox Inc.
mahmoud.abokhamis@logicblox.com

Hung Q. Ngo
LogicBlox Inc.
hung.ngo@logicblox.com

Atri Rudra
University at Buffalo, SUNY
atri@buffalo.edu

ABSTRACT

We define and study the **F**unctional **A**ggregate **Q**uery (FAQ) problem, which captures common computational tasks across a very wide range of domains including relational databases, logic, matrix and tensor computation, probabilistic graphical models, constraint satisfaction, and signal processing. Simply put, an FAQ is a declarative way of defining a new function from a database of input functions.

We present InsideOut, a dynamic programming algorithm, to evaluate an FAQ. The algorithm rewrites the input query into a set of easier-to-compute FAQ sub-queries. Each subquery is then evaluated using a worst-case optimal relational join algorithm. The topic of designing algorithms to optimally evaluate the classic multiway join problem has seen exciting developments in the past few years. Our framework tightly connects these new ideas in database theory with a vast number of application areas in a coherent manner, showing potentially that – with the right abstraction, blurring the distinction between data and computation – a good database engine can be a general purpose constraint solver, relational data store, graphical model inference engine, and matrix/tensor computation processor all at once.

The InsideOut algorithm is very simple, as shall be described in this paper. Yet, in spite of solving an extremely general problem, its runtime either is as good as or improves upon the best known algorithm for the applications that FAQ specializes to. These corollaries include computational tasks in graphical model inference, matrix/tensor operations, relational joins, and logic. Better yet, InsideOut can be used within any database engine, because it is basically a principled way of rewriting queries. Indeed, it is already part of the LogicBlox database engine, helping efficiently answer traditional database queries, graphical model inference queries, and train a large class of machine learning models inside the database itself.

This work was partly supported by NSF grant CCF-1319402 and by DARPA under agreement #FA8750-15-2-0009. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright thereon.

1. INTRODUCTION

The following fundamental problems from diverse domains share a common algebraic structure involving (generalized) sums of products.

Example 1. (Matrix Chain Multiplication (MCM)) Given a series of matrices $\mathbf{A}_1, \ldots, \mathbf{A}_n$ over some field $\mathbb{F}$, where the dimension of $\mathbf{A}_i$ is $p_i \times p_{i+1}$, $i \in [n]$, we wish to compute the product $\mathbf{A} = \mathbf{A}_1 \cdots \mathbf{A}_n$. The problem can be reformulated as follows. There are $n+1$ variables $X_1, \ldots, X_{n+1}$ with domains $\mathsf{Dom}(X_i) = [p_i]$, for $i \in [n+1]$. For $i \in [n]$, matrix $\mathbf{A}_i$ can be viewed as a function of two variables

$$\psi_{i,i+1} : \mathsf{Dom}(X_i) \times \mathsf{Dom}(X_{i+1}) \to \mathbb{F},$$

where $\psi_{i,i+1}(x, y) = (\mathbf{A}_i)_{xy}$. The MCM problem is to compute the output function

$$\varphi(x_1, x_{n+1}) = \sum_{x_2 \in \mathsf{Dom}(X_2)} \cdots \sum_{x_n \in \mathsf{Dom}(X_n)} \prod_{i=1}^{n} \psi_{i,i+1}(x_i, x_{i+1}).$$

Example 2. (Maximum A Posteriori (MAP) queries in probabilistic graphical models (PGMs)) Consider a discrete graphical model represented by a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$. There are n discrete random variables $\mathcal{V} = \{X_1, \ldots, X_n\}$ on finite domains $\mathsf{Dom}(X_i)$, $i \in [n]$, and $m = |\mathcal{E}|$ *factors*

$$\psi_S : \prod_{i \in S} \mathsf{Dom}(X_i) \to \mathbb{R}_+, \ S \in \mathcal{E}.$$

A typical inference task is to compute the marginal MAP estimates, written in the form

$$\varphi(x_1, \ldots, x_f) = \max_{x_{f+1} \in \mathsf{Dom}(X_{f+1})} \cdots \max_{x_n \in \mathsf{Dom}(X_n)} \prod_{S \in \mathcal{E}} \psi_S(\mathbf{x}_S).$$

Example 3. (Conjunctive query in RDBMS) Consider a schema with the following input relations: $R(a, b)$, $S(b, c)$, $T(c, a)$, where for simplicity let us say all attributes are integers. Consider the following query:

```
SELECT R.a
FROM R, S, T
WHERE R.b = S.b AND S.c = T.c AND T.a = R.a;
```

The above query can be reformulated as follows. Relation $R(a, b)$ is modeled by a function $\psi_R(a, b) \to \{\mathsf{true}, \mathsf{false}\}$,

where $\psi_R(a,b) = \mathsf{true}$ iff $(a,b) \in R$, and relations $S(b,c)$ and $T(c,a)$ are modeled by similar functions $\psi_S(b,c), \psi_T(c,a)$. Now, computing the above query basically corresponds to computing the function $\varphi(a) \to \{\mathsf{true}, \mathsf{false}\}$, defined as:

$$\varphi(a) = \bigvee_b \bigvee_c \psi_R(a,b) \wedge \psi_S(b,c) \wedge \psi_T(c,a).$$

Example 4. (# Quantified Conjunctive Query (#QCQ)) Let Φ be a first-order formula of the form

$$\Phi(X_1, \ldots, X_f) = Q_{f+1} X_{f+1} \cdots Q_n X_n \left(\bigwedge_{R \in \mathsf{atoms}(\Phi)} R \right),$$

where $Q_i \in \{\exists, \forall\}$, for $i > f$. The #QCQ problem is to *count* the number of tuples in relation Φ on the free variables $X_1, \ldots, X_f$. To reformulate #QCQ, construct a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ as follows: $\mathcal{V}$ is the set of all variables $X_1, \ldots, X_n$, and for each $R \in \mathsf{atoms}(\Phi)$ there is a hyperedge $S = \mathsf{vars}(R)$ consisting of all variables in R. The atom R can be viewed as a function indicating whether an assignment $\mathbf{x}_S$ to its variables is satisfied by the atom; namely $\psi_S(\mathbf{x}_S) = 1$ if $\mathbf{x}_S \in R$ and 0 otherwise.

For each $i \in \{f+1, \ldots, n\}$ we define an aggregate operator

$$\bigoplus^{(i)} = \begin{cases} \max & \text{if } Q_i = \exists, \\ \times & \text{if } Q_i = \forall. \end{cases}$$

Then, the #QCQ problem above is to compute the *constant* function

$$\varphi = \sum_{x_1 \in \mathsf{Dom}(X_1)} \cdots \sum_{x_f \in \mathsf{Dom}(X_f)} \bigoplus_{x_{f+1} \in \{0,1\}}^{(f+1)} \cdots \bigoplus_{x_n \in \{0,1\}}^{(n)} \prod_{S \in \mathcal{E}} \psi_S(\mathbf{x}_S).$$

It turns out that these and dozens of other fundamental problems from constraint satisfaction (CSP), databases, matrix operations, PGM inference, logic, coding theory, and complexity theory can be viewed as special instances of a generic problem we call the **F**unctional **A**ggregate **Q**uery, or the FAQ problem, which we define next. (See [2, 6] for many more examples.)

Throughout the paper, we use the following convention. Uppercase X_i denotes a variable, and lowercase x_i denotes a value in the domain $\mathsf{Dom}(X_i)$ of the variable. Furthermore, for any subset $S \subseteq [n]$, define

$$\mathbf{X}_S = (X_i)_{i \in S}, \qquad \mathbf{x}_S = (x_i)_{i \in S} \in \prod_{i \in S} \mathsf{Dom}(X_i).$$

In particular, $\mathbf{X}_S$ is a tuple of variables and $\mathbf{x}_S$ is a tuple of specific values with support S. The input to FAQ is a set of functions and the output is a function computed using a series of aggregates over the variables and input functions. More specifically, for each $i \in [n]$, let X_i be a variable on some discrete domain $\mathsf{Dom}(X_i)$, where $|\mathsf{Dom}(X_i)| \geq 2$. The FAQ problem is to compute the following function

$$\varphi(\mathbf{x}_{[f]}) = \bigoplus_{x_{f+1} \in \mathsf{Dom}(X_{f+1})}^{(f+1)} \cdots \bigoplus_{x_n \in \mathsf{Dom}(X_n)}^{(n)} \bigotimes_{S \in \mathcal{E}} \psi_S(\mathbf{x}_S), \quad (1)$$

where

- $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ is a multi-hypergraph. $\mathcal{V} = [n]$ is the index set of the variables X_i, $i \in [n]$. Overloading notation, $\mathcal{V}$ is also referred to as the set of variables.
- The set $F = [f]$ is the set of *free variables* for some integer $0 \leq f \leq n$. Variables in $\mathcal{V} - F$ are called *bound variables*. (Free and bound are logic terminologies. Free variables are group-by variables in database nomenclature.)
- $\mathbf{D}$ is a fixed domain, such as $\{0,1\}$, $\mathbb{R}^+$, $\mathbb{Z}$.
- For every hyperedge $S \in \mathcal{E}$, $\psi_S : \prod_{i \in S} \mathsf{Dom}(X_i) \to \mathbf{D}$ is an *input* function (also called a *factor*).
- For every bound variable $i > f$, $\oplus^{(i)}$ is a binary (aggregate) operator on the domain $\mathbf{D}$.
- And, for each bound variable $i > f$ either $\oplus^{(i)} = \otimes$ or $(\mathbf{D}, \oplus^{(i)}, \otimes)$ forms a commutative semiring[1] (with the same $\mathbf{0}$ and $\mathbf{1}$). Informally, this means that we can do addition and multiplication over $\mathbf{D}$ and still remain in the same set.

  If $\oplus^{(i)} = \otimes$, then $\oplus^{(i)}$ is called a *product aggregate*; otherwise, it is a *semiring aggregate*. (We assume that there is at least one semiring aggregate.)

Because for $i > f$ every variable X_i has its own aggregate $\bigoplus^{(i)}$ over all values $x_i \in \mathsf{Dom}(X_i)$, in the rest of the paper we will write $\bigoplus_{x_i}^{(i)}$ to mean $\bigoplus_{x_i \in \mathsf{Dom}(X_i)}^{(i)}$.

We will refer to φ as an FAQ-*query*. We use FAQ-SS to denote the special case when there is a *S*ingle *S*emiring aggregate, i.e. $\oplus^{(i)} = \oplus, \forall i > f$, and $(\mathbf{D}, \oplus, \otimes)$ is a semiring [6].

Example 5. (Aggregate query in RDBMS) Consider the following query over relations $R(a,b)$, $S(a,c)$, $T(b,c,d,e)$, $U(d,f)$, $V(e,f)$, $W(e,g)$, $Y(f,h)$, where all attributes are integers:

```
SELECT R.b, U.d, sum(W.e)
FROM R, S, T, U, V, W, Y
WHERE R.a = S.a AND R.b = T.b AND S.c = T.c
      AND T.d = U.d AND T.e = V.e AND W.e = V.e
      AND U.f = V.f AND Y.f = V.f GROUP BY R.b, U.d;
```

We now explain how the above query can be reduced to an FAQ instance. Relation $R(a,b)$ is modeled with a function $\psi_R(a,b) \to \{0,1\}$, where $\psi_R(a,b) = 1$ iff $(a,b) \in R$. Similarly, we can think of relations S, T, U, V, and Y as functions ψ_S, ψ_T, ψ_U, ψ_V, ψ_Y, with $\{0,1\}$ values. We single out one relation $W(e,g)$ where the modeling is different: $\psi_W(e,g) = e$ if $(e,g) \in W$ and 0 otherwise. The corresponding FAQ-query is

$$\varphi(b,d) = \sum_a \sum_c \sum_e \sum_f \sum_g \sum_h \psi_R \psi_S \psi_T \psi_U \psi_V \psi_W \psi_Y$$

(For readability, we did not write the argument lists of the functions ψ_R, ψ_S, etc. They should be obvious from context.) Note that a tuple in the output of the aggregate query has the schema $(b, d, \varphi(b,d))$. The corresponding hypergraph is shown in Fig. 1a. The set of free variables is $F = \{b, d\}$. The domain is $\mathbf{D} = \mathbb{Z}$, the set of integers. Note

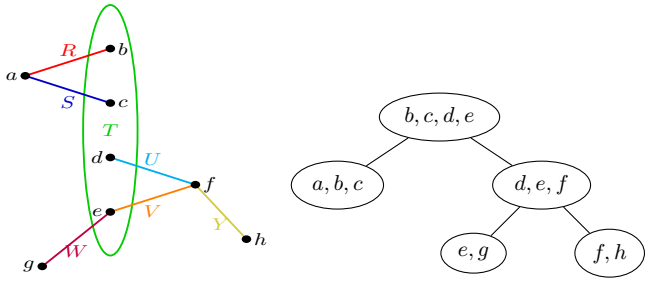
[1] A triple $(\mathbf{D}, \oplus, \otimes)$ is a *commutative semiring* if $\oplus$ and $\otimes$ are commutative binary operators over $\mathbf{D}$ satisfying the following: (1) $(\mathbf{D}, \oplus)$ is a commutative monoid with an additive identity, denoted by $\mathbf{0}$. (2) $(\mathbf{D}, \otimes)$ is a commutative monoid with a multiplicative identity, denoted by $\mathbf{1}$. (3) $\otimes$ distributes over $\oplus$. (4) For any element $e \in \mathbf{D}$, $e \otimes \mathbf{0} = \mathbf{0} \otimes e = \mathbf{0}$.

also that the above reduction to FAQ still works if we replace sum by another aggregate, e.g., max.

In order to explain later the connection of InsideOut to query rewriting, we also write the above query in LogiQL, an extension of Datalog supported by the LogicBlox engine [7]:

```
Q[b, d] = s <- agg<<s = total(e)>> R(a,b), S(a,c),
    T(b,c,d,e), U(d,f), V(e,f), W(e,g), Y(f,h).
```

In the above, `agg` is short for aggregate, `total` is equivalent to `sum` in SQL, the notation `Q[b,d]=s` means that the head predicate is `Q(b,d,s)` where `(b,d)` is a key, hence the query computes `Q(b,d,sum(e))`.



(a) Query hypergraph (b) Tree decomposition

Figure 1: Query from Example 5

The above example illustrates several important points. First, when we defined the FAQ problem we did not specify how the input and output factors are represented. The representation choice turns out to make a huge difference in computational complexity [2]. However, in practical applications the representation is usually the obvious one: an input factor $\psi_S(\mathbf{X}_S)$ can be thought of as a table of tuples $[\mathbf{x}_S, \psi_S(\mathbf{x}_S)]$, with the implicit assumption that if $\mathbf{x}_S$ is not in the table then its ψ_S-value is $\mathbf{0}$. (This is the additive identity $\mathbf{0}$ of the domain $\mathbf{D}$.) Second, the reduction to FAQ is only at the syntax level. No real data conversion is necessary. All the data we need to obtain the functions ψ_R, ψ_T etc. are already in the input relations. Third, in the mathematical definition of $\varphi(b,d)$ above, the domains of all variables are integers and so we have *infinite* sums. We could have restricted all variables to their active domains; but that is not necessary because summing over all integers or over the active domains give identical answer: tuples not present are assumed to have values $\mathbf{0}$.

Now that we have established the scope of FAQ, in the remainder of this paper we show a perhaps surprising result that an FAQ problem can be solved by *one* simple yet efficient algorithm. The algorithm can be implemented as a set of ordinary database queries. The runtime matches or improves upon the best known runtimes in many application areas that the FAQ framework captures. The runtime depends on the order of variable aggregates in the FAQ expression, which naturally leads us to the question of how to re-order those aggregates to obtain the best runtime without changing the semantic meaning of the expression.

2. THE INSIDEOUT ALGORITHM

Parts of this section will be familiar to readers who have been exposed to elementary graphical models [24]. There are, however, a couple of ideas that are taken from new developments in database theory [29, 28, 3] that are likely not known in the graphical model literature. For each factor ψ_S, define its *size* to be the number of non-zero points under its domain: $|\psi_S| := \left|\left\{\mathbf{x}_S \mid \psi_S(\mathbf{x}_S) \neq \mathbf{0}\right\}\right|$.

Basic variable elimination. To describe the intuition, we first explain InsideOut as it applies to the special case of FAQ-SS (or SumProd). The idea behind variable elimination [17, 37, 36] is to 'fold' common factors, exploiting the distributive law:

$$\bigoplus_{x_{f+1}} \cdots \bigoplus_{x_n} \bigotimes_{S\in\mathcal{E}} \psi_S(\mathbf{x}_S)$$

$$= \bigoplus_{x_{f+1}} \cdots \bigoplus_{x_{n-1}} \bigotimes_{S\in\mathcal{E}-\partial(n)} \psi_S(\mathbf{x}_S) \otimes \underbrace{\left(\bigoplus_{x_n} \bigotimes_{S\in\partial(n)} \psi_S(\mathbf{x}_S)\right)}_{\text{new factor } \psi_{U_n-\{n\}}},$$

where the equality follows from the fact that $\otimes$ distributes over $\oplus$, $\partial(n)$ denotes all edges incident to n in $\mathcal{H}$ and $U_n = \cup_{S\in\partial(n)} S$. Assume for now that we can somehow efficiently compute the intermediate factor $\psi_{U_n-\{n\}}$. Then, the resulting problem is another instance of FAQ-SS on a modified multi-hypergraph $\mathcal{H}'$, constructed from $\mathcal{H}$ by removing vertex n along with all edges in $\partial(n)$, and *adding back* a new hyperedge $U_n - \{n\}$. Recursively, we continue this process until all variables $X_n, \ldots, X_{f+1}$ are eliminated. Textbook treewidth-based results for PGM inference are obtained this way [24]. In the database context (i.e. given an FAQ-query over the Boolean semiring), the intermediate result $\psi_{U_n-\{n\}}$ is essentially an intermediate relation of a query plan, the folding technique exploiting distributive law corresponds to "pushing the aggregate down" the query plan [13].

Introducing the indicator projections. While correct, basic variable elimination as described above is potentially not very efficient for sparse input factors, i.e. factors where the number of non-zero entries is much smaller than the product of the active domain sizes. This is because the product that was factored out of the scope of X_n might annihilate many entries of the intermediate result $\psi_{U_n-\{n\}}$, while we have spent so much time computing $\psi_{U_n-\{n\}}$. For example, for an $S \notin \partial(n)$ such that $S \subseteq U_n$ and tuple $\mathbf{y}_S$ such that $\psi_S(\mathbf{y}_S) = \mathbf{0}$, we do not need to compute the entries $\psi_{U_n-\{n\}}(\mathbf{x}_{U_n-\{n\}})$ for which $\mathbf{y}_S = \mathbf{x}_S$: those entries will be eliminated later anyhow. The idea is then to only compute those $\psi_{U_n-\{n\}}(\mathbf{x}_{U_n-\{n\}})$ values that will "survive" the other factors later on. One simple way to achieve this would be to compute, for each $S \in \mathcal{E} - \partial(n)$, an "indicator factor" that checks if $\psi_S(\mathbf{x}_S)$ is $\mathbf{0}$ or not. Formally, for any two sets $T \subseteq S$, and a given factor ψ_S, the function $\psi_{S/T} : \prod_{i\in T} \mathsf{Dom}(X_i) \to \mathbf{D}$ defined by

$$\psi_{S/T}(\mathbf{x}_T) := \begin{cases} \mathbf{1} & \exists \mathbf{x}_{S-T} \text{ s.t. } \psi_S(\mathbf{x}_T, \mathbf{x}_{S-T}) \neq \mathbf{0} \\ \mathbf{0} & \text{otherwise} \end{cases}$$

is called the *indicator projection* of ψ_S onto T. Using indicator factors, InsideOut computes the following factor when marginalizing X_n away:

$$\psi_{U_n-\{n\}}(\mathbf{x}_{U_n-\{n\}}) = \bigoplus_{x_n} \left[\left(\bigotimes_{S\in\partial(n)} \psi_S\right) \otimes \left(\bigotimes_{\substack{S\notin\partial(n),\\ S\cap U_n\neq\emptyset}} \psi_{S/S\cap U_n}\right)\right]. \quad (2)$$

Another minor tweak is the observation that, if there is a hyperedge $S \in \mathcal{E} - \partial(n)$ for which $S \subset U_n$, then we do not use the indicator projection $\psi_{S/S\cap U_n}$: we can use ψ_S itself to compute the intermediate factor $\psi_{U_n-\{n\}}$, and then remove ψ_S from $\mathcal{H}'$.

Example 6. We explain how the ideas above are implemented in Example 5. First, the order in which we choose to eliminate variables might have a huge effect on the runtime. For now, let us assume that we somehow decided to rewrite $\varphi(b,d)$ using the following variable order, where we trace the first couple of steps of the InsideOut algorithm *without* the indicator projection: (Example 10 later explains how this order is related to the tree decomposition in Fig. 1b.)

$$\begin{aligned}
\varphi(b,d) &= \sum_c \sum_a \sum_e \sum_f \sum_g \sum_h \psi_R\psi_S\psi_T\psi_U\psi_V\psi_W\psi_Y \\
&= \sum_c \sum_a \sum_e \sum_f \sum_g \psi_R\psi_S\psi_T\psi_U\psi_V\psi_W \underbrace{\sum_h \psi_Y(f,h)}_{\psi_1(f)} \\
&= \sum_c \sum_a \sum_e \sum_f \sum_g \psi_R\psi_S\psi_T\psi_U\psi_V\psi_W\psi_1 \\
&= \sum_c \sum_a \sum_e \sum_f \psi_R\psi_S\psi_T\psi_U\psi_V\psi_1 \underbrace{\sum_g \psi_W(e,g)}_{\psi_2(e)} \\
&= \sum_c \sum_a \sum_e \sum_f \psi_R\psi_S\psi_T\psi_U\psi_V\psi_1\psi_2
\end{aligned}$$

The first two steps are straightforward, where we eliminated g and h. In LogiQL, these intermediate factors are computed with the following two rules

```
psi1[f] = s1 <- agg<<s1 = count()>> Y(f,h).
psi2[e] = s2 <- agg<<s2 = total(e)>> W(e,g).
```

The mathematical abstraction corresponds to rewriting a query into a series of smaller queries. Next, we explain how the indicator projection works when we eliminate variable f. Out of the remaining factors ψ_R,ψ_S,ψ_T,ψ_U,ψ_V,ψ_1,and ψ_2,the following factors contain f: $\psi_U(d,f)$ $\psi_V(e,f)$ and $\psi_1(f)$. If we were to multiply them together and marginalize away f, we would create a new factor $\psi_3(e,d) = \sum_f \psi_U\psi_V\psi_1$ over variables $\{e,d\}$. However, two other factors have variables that overlap with $\{e,d\}$, namely $\psi_T(b,c,d,e)$ and $\psi_2(e)$. For ψ_T, we include its indicator projection $\psi_{T/\{e,d\}}$ in computing ψ_3. (We will see later in Example 7 how including $\psi_{T/\{e,d\}}$ can actually speed up the computation of ψ_3 asymptotically.) For ψ_2, we can include ψ_2 itself. (Recall the minor tweak we mentioned above.) Overall, we end up with the following definition of ψ_3:

$$\psi_3(e,d) = \sum_f \psi_U \cdot \psi_V \cdot \psi_1 \cdot \psi_2 \cdot \psi_{T/\{e,d\}}.$$

In LogiQL, this sub-result is computed with two rules:

```
proj1(d,e) <- T(b,c,d,e). // projection rule
psi3[e,d] = s3 <- agg<<s3 = total(s1*s2)>> U(d,f),
    V(e,f), psi1[f] = s1, psi2[e] = s2, proj1(d,e).
```

After eliminating f, we are left with the following

$$\begin{aligned}
\varphi(b,d) &= \sum_c \sum_a \sum_e \psi_R\psi_S\psi_T\psi_3 \\
&= \sum_c \sum_a \psi_R\psi_S \sum_e \psi_3\psi_T \\
&= \sum_c \sum_a \psi_R\psi_S \underbrace{\sum_e \psi_3\psi_T\psi_{R/\{b\}}\psi_{S/\{c\}}}_{\psi_4(b,c,d)}
\end{aligned}$$

leading to the following LogiQL rules

```
proj2(b) <- R(a,b).
proj3(c) <- S(a,c).
psi4[b,c,d] = s4 <- agg<<s4 = total(s3)>>
   psi3[e,d] = s3, T(b,c,d,e), proj2(b), proj3(c).
```

At this point, we have 3 factors left $\psi_R(a,b)$, $\psi_S(a,c)$, and $\psi_4(b,c,d)$. We eliminate a then c straightforwardly:

$$\sum_c \sum_a \psi_R\psi_S\psi_4 = \sum_c \psi_4 \underbrace{\sum_a \psi_R\psi_S\psi_{4/\{b,c\}}}_{\psi 5(b,c)} = \sum_c \psi_4\psi_5.$$

Note that $\psi_{4/\{b,c\}}$ has values in $\{0,1\}$ although ψ_4 can have any value in $\mathbb{Z}$. The final LogiQL rules are

```
proj4(b,c) <- psi4[b,c,d] = s4. // indicator projection
psi5[b,c] = s5 <- agg<<s5 = count()>>
                  R(a,b), S(a,c), proj4(b,c).
output[b,d] = t <- agg<<t = total(s4*s5)>>
                  psi4[b,c,d] = s4, psi5[b,c] = s5.
```

The general FAQ problem. The above strategy does not care if the variable aggregates where the same or different: As long as $(\mathbf{D}, \oplus^{(n)}, \otimes)$ is a semiring, we can fold the common factors and eliminate X_n. Thus, InsideOut works almost as is for a general FAQ instance (as opposed to FAQ-SS). Finally, when $\oplus^{(n)} = \otimes$ we simply swap the two (identical) operators:

$$\begin{aligned}
\varphi(\mathbf{x}_{[f]}) &= \cdots \bigoplus_{x_{n-1}}^{(n-1)} \bigoplus_{x_n}^{(n)} \bigotimes_{S\in\mathcal{E}} \psi_S(\mathbf{x}_S) \\
&= \cdots \bigoplus_{x_{n-1}}^{(n-1)} \bigotimes_{x_n\in \mathsf{Dom}(X_n)} \bigotimes_{S\in\mathcal{E}} \psi_S(\mathbf{x}_S) \\
&= \cdots \bigoplus_{x_{n-1}}^{(n-1)} \bigotimes_{S\in\mathcal{E}} \bigotimes_{x_n\in \mathsf{Dom}(X_n)} \psi_S(\mathbf{x}_S) \\
&= \cdots \bigoplus_{x_{n-1}}^{(n-1)} \bigotimes_{S\notin\partial(n)} \underbrace{(\psi_S(\mathbf{x}_S))^{|\mathsf{Dom}(X_n)|}}_{\psi'_S} \bigotimes_{S\in\partial(n)} \underbrace{\bigotimes_{x_n} \psi_S(\mathbf{x}_S)}_{\psi_{S-\{n\}}}.
\end{aligned}$$

We are left with an FAQ-instance whose hypergraph is exactly $\mathcal{H}' = \mathcal{H} - \{n\}$: the hypergraph obtained form $\mathcal{H}$ by removing vertex n from the vertex set and all incident hyperedges. The sub-problems are of the form of *product marginalizations* of individual factors ψ_S for $S \in \partial(n)$, each of which can be computed in linear time in $|\psi_S|$. The product marginalization step is algorithmically much easier because it does not create the intermediate factor $\psi_{U_n-\{n\}}$. As for $S \notin \partial(n)$, we replace ψ_S by the power factor $\psi'_S(\mathbf{x}_S) = (\psi_S(\mathbf{x}_S))^{|\mathsf{Dom}(X_n)|}$, which can be done in linear time with a $\log|\mathsf{Dom}(X_n)|$ blowup using the repeated squaring algorithm. Note the key fact that this power is with respect to the product aggregate $\otimes$. In most (if not all) applications of FAQ, there is one additional property: most of the time, $\otimes$ is an idempotent operator over the active domain. For example, in the #QCQ problem $\otimes$ is the usual product operator and the domain that it aggregates over is $\{0,1\}$ (before there is a sum outside). In this case, $\psi'_S(\mathbf{x}_S) =$

$(\psi_S(\mathbf{x}_S))^{|\mathsf{Dom}(X_n)|} = \psi_S(\mathbf{x}_S)$, and we do not need to spend the linear nor log-blowup time. For more details on product idempotence, see [2].

FAQ sub-problems as natural joins. In the above we have explained how InsideOut breaks a big problem into smaller problems. In the product marginalization case, the sub-problems are easy to solve: they can be solved in linear time. The most difficult problems, however, are of the form (2). This is exactly an FAQ-query where we marginalize out only one variable, with the remaining variables free. Zooming in, problem (2) is of the form

$$\psi_{U_n-\{n\}}(\mathbf{x}_{U_n-\{n\}}) := \bigoplus_{x_n} \bigotimes_{F\in\mathcal{E}_n} \psi_F,$$

where $\mathcal{H}_n = (U_n, \mathcal{E}_n)$ is the sub-FAQ-query hypergraph. The problem is solved by computing $\psi_{U_n}(\mathbf{x}_{U_n}) := \bigotimes_{F\in\mathcal{E}_n} \psi_F$ first. Once the ψ_{U_n} is computed, marginalizing away X_n to obtain $\psi_{U_n-\{n\}}$ is trivial.

Computing the inner product is a natural join problem in disguise. Each input factor ψ_S is represented using a table of tuples of the form $[\mathbf{x}_S, \psi_S(\mathbf{x}_S)]$. Essentially, $\mathbf{x}_S$ is the (compound) key and $\psi_S(\mathbf{x}_S)$ is the value in this relation. Again, recall that entries not in the table have ψ_S-value $\mathbf{0}$. Hence, to compute ψ_{U_n} we can first join the tables ψ_S using only the key space. For each tuple $\mathbf{x}_{U_n}$ in the result of this join, we record the value $\psi_{U_n}(\mathbf{x}) = \prod_{F\in\mathcal{E}_n} \psi_F(\mathbf{x}_F)$. The runtime is dominated by the natural join's runtime.

Worst-case optimal join algorithms. Computing the natural join is a very well-studied problem with exciting new developments in the past decade or so. There are new *worst-case optimal* algorithms [35, 28, 29, 1] that operate quite differently from traditional query plans, in the sense that they no longer compute one pairwise join at a time, but instead process the query globally. While the vast majority of database engines today still rely on traditional query plans, new, complex data analytics engines are switching to worst-case optimal algorithms: LogicBlox's engine [7] is built on a worst-case optimal algorithm called LeapFrog Triejoin [35] (LFTJ), and the *Myria* data analytics platform supports a variant of LFTJ [12].

We briefly outline these results here. The generic form of the natural join problem can be posed in our hypergraph language as $Q = \bowtie_{F\in\mathcal{E}} R_F$, where $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ is the query hypergraph. The vertices of this hypergraph consist of all attributes. Each hyperedge $F \in \mathcal{E}$ corresponds to an input relation R_F whose attributes are F. The natural join problem can be thought of as a constraint satisfaction problem: each input relation R_F imposes a constraint where a tuple $\mathbf{x}_F$ satisfies the constraint if $\mathbf{x}_F \in R_F$. A tuple $\mathbf{x}$ on all variables $\mathcal{V}$ is an output of the join if the projection $\mathbf{x}_F$ satisfies R_F for all $F \in \mathcal{E}$.

LFTJ [35] can be viewed as *backtracking-search* algorithm, which was known some 50 years ago in the AI and constraint programming world [16, 20]. (In contrast, by saving intermediate results $\psi_{U_n-\{n\}}$ instead of re-computing them each time, InsideOut can be thought of as *dynamic programming*. The duality between backtracking search and dynamic programming is well-known [33].) LFTJ fixes some variable ordering $X_1, \ldots, X_n$ of the query Q, then performs "leap-frogging" to find the first binding x_1 that does not yet violate any constraints R_F; once x_1 is found, it looks for the first binding x_2 such that the partial tuple (x_1, x_2) does not violate any constraint. The algorithm proceeds this way until either a full binding $\mathbf{x}$ is constructed in which case $\mathbf{x}$ is an output, or no good binding is found. For example, if no feasible binding for x_3 is found, then the algorithm backtracks to the next good binding of x_2.

The first advantage of backtracking search is that it requires only $O(1)$-extra space: it does not cache any computation. The second advantage, amazingly, is that a join algorithm based on back-tracking search such as LFTJ or others in [28, 29] are worst-case optimal, in the sense that the algorithm runs in time bounded by the worst-case output size. To state the output size bound, we need the following notion. Define the *fractional edge cover polytope* $\mathsf{P}(\mathcal{H})$ associated with a hypergraph $\mathcal{H}$ to be the set of all vectors $\boldsymbol{\lambda} = (\lambda_F)_{F\in\mathcal{E}}$ satisfying the following linear constraints:

$$\boldsymbol{\lambda} \geq \mathbf{0}, \text{ and } \sum_{F\in\mathcal{E}, v\in F} \lambda_F \geq 1, \ \forall v \in \mathcal{V}.$$

A vector $\boldsymbol{\lambda} \in \mathsf{P}(\mathcal{H})$ is called a *fractional edge cover* of $\mathcal{H}$. The join output size is bounded above by $\prod_{F\in\mathcal{E}} |R_F|^{\lambda_F}$, for *any* $\boldsymbol{\lambda} \in \mathsf{P}(\mathcal{H})$. The best bound $\mathsf{AGM}(\mathcal{H})$, known as the AGM-bound [8, 21], is obtained by solving the linear program

$$\min\Big\{ \sum_{F\in\mathcal{E}} \lambda_F \log_2 |R_F| : \boldsymbol{\lambda} \in \mathsf{P}(\mathcal{H}) \Big\}. \tag{3}$$

Example 7. Consider the query computing ψ_3 in Example 6. The join query on the keys has the following shape: $Q = U(d,f) \bowtie V(e,f) \bowtie I(f) \bowtie J(e) \bowtie K(d,e)$. Then, $\mathsf{AGM}(Q) = |U|^{\lambda_{d,f}} |V|^{\lambda_{e,f}} |I|^{\lambda_f} |J|^{\lambda_e} |K|^{\lambda_{d,e}}$, where $\boldsymbol{\lambda}$ is a fractional edge cover of the query's hypergraph. Suppose all input relations have the same size N, then the optimal bound is obtained by setting $\lambda_{d,f} = \lambda_{d,e} = \lambda_{e,f} = 1/2$, and $\lambda_d = \lambda_e = 0$. Worst-case optimal algorithms run in time $\tilde{O}(N^{3/2})$ for this instance. Any traditional join-tree based plan runs in $\Omega(N^2)$-time for some input [28]. Moreover, without the indicator projection of $T(b,c,d,e)$, there would be no $K(d,e)$ above, the best edge cover would be $\lambda_{d,f} = \lambda_{e,f} = 1$, and the runtime would become $\Omega(N^2)$.

Runtime analysis. Let N denote the input size, $|\mathsf{output}|$ the output size, and K the set of $k \in [n]$ for which $\oplus^{(k)} \neq \otimes$ (note that $[f] \subseteq K$). Also, let $\mathsf{AGM}(Q_k)$ denote the AGM-bound on the kth sub-query's hypergraph $\mathcal{H}_k$. Then, it is not hard to show [2] that the runtime of InsideOut is

$$\tilde{O}\Big(N + \sum_{k\in K} \mathsf{AGM}(\mathcal{H}_k) + |\mathsf{output}|\Big). \tag{4}$$

The first term is input-preprocessing time, second is the total subproblem solving time, and third is the unavoidable output reporting time. From (4), we can write down a precise expression for the runtime of InsideOut. Minimizing the resulting (somewhat complicated) expression leads to the dynamic programming algorithm for the MCM problem and the FFT algorithm for the DFT (see [2] for details).

In the above discussion, we assumed that variables were eliminated in order $X_n, X_{n-1}, \ldots, X_1$. However, there is no reason to force InsideOut to follow this particular order. In particular, there might be a different variable ordering for which expression (4) is *a lot* smaller and the algorithm still works correctly on that ordering (see [2]). This is where the main technical contributions of our work in [2] begin. We need to answer the following two fundamental questions:

Question 1. How do we know which variable orderings are equivalent to the original FAQ-query expression?

Question 2. How do we find the "best" variable ordering among all equivalent variable orderings?

In the next two sections, we sketch how we answered the above two questions and followup questions in theory and in practice.

3. THEORETICAL CONTRIBUTIONS

To answer the above questions, we start with some definitions. A variable ordering σ is φ-*equivalent* iff permuting the variable aggregates of φ using σ gives an expression φ' that is *semantically-equivalent* to φ, i.e. that always returns the same output as φ *no matter* what the input is. Let $\mathsf{EVO}(\varphi)$ denote the set of all φ-equivalent variable orderings.

Example 8. The FAQ query φ' below is φ-equivalent.

$$\begin{aligned}\varphi &= \sum_a \sum_d \max_b \sum_c \psi_1(a,b)\psi_2(a,c)\psi_3(c,d),\\ \varphi' &= \sum_a \sum_c \sum_d \max_b \psi_1(a,b)\psi_2(a,c)\psi_3(c,d).\end{aligned}$$

This is because φ can be written as

$$\varphi = \sum_a \left[\left(\sum_d \sum_c \psi_2(a,c)\psi_3(c,d) \right) \left(\max_b \psi_1(a,b) \right) \right].$$

Now, for any $\sigma \in \mathsf{EVO}(\varphi)$, let $\mathcal{H}_k^\sigma$ denote the kth subquery's hypergraph when we run InsideOut on σ. Ideally, we would like to find σ minimizing the expression $\sum_{k\in K} \mathsf{AGM}(\mathcal{H}_k^\sigma)$. However, this expression is data-dependent and thus it is a bit difficult to handle in a mathematically clean way. We simplify our objective by approximating the bound (4) a little: we upperbound $\mathsf{AGM}(\mathcal{H}_k^\sigma)$ by the *fractional edge cover number* of the subgraph $\mathcal{H}_k^\sigma$, i.e. $\mathsf{AGM}(\mathcal{H}_k^\sigma) \le N^{\rho^*(\mathcal{H}_k^\sigma)}$, where $\rho^*(\mathcal{H}_k^\sigma) := \min\{\sum_{F\in\mathcal{E}} \lambda_F : \boldsymbol{\lambda} \in \mathsf{P}(\mathcal{H}_k^\sigma)\}$. Then, (4) is upperbounded by $\tilde{O}\left(N^{\max_{k\in K} \rho^*(\mathcal{H}_k^\sigma)} + |\mathsf{output}|\right)$; InsideOut on variable ordering σ runs in $\tilde{O}(N^{\mathsf{faqw}(\sigma)} + |\mathsf{output}|)$-time, where $\mathsf{faqw}(\sigma) := \max_{k\in K} \rho^*(\mathcal{H}_k^\sigma)$. Thus, to have the best runtime we would like to select an equivalent ordering σ with the smallest exponent $\mathsf{faqw}(\sigma)$, called the FAQ-*width* of an FAQ-query:

$$\mathsf{faqw}(\varphi) := \min\{\mathsf{faqw}(\sigma) \mid \sigma \in \mathsf{EVO}(\varphi)\} \tag{5}$$

Example 9. In Example 8, the original order in φ has an FAQ-width of 2, because eliminating c first corresponds to joining ψ_2 and ψ_3 in time $\Omega(N^2)$. In contrast, the order in φ' has an faqw of 1, allowing to evaluate φ in time $O(N)$.

To solve the optimization problem (5), the first problem we have to address is to precisely characterize the set $\mathsf{EVO}(\varphi)$. Our approach, sketched in Fig. 2, is to construct an *expression tree* of the FAQ query φ. The expression tree defines a partially ordered set on the variables called the *precedence poset*. Then, to complete the characterization of EVO, we show that every ordering in EVO is *component-wise equivalent* (CWE) to a linear extension of $\mathsf{LinEx}(P)$. (See [2] for details.) Thus, if we do not care about query complexity, we can take the orange path in Fig 2 and bruteforcedly compute an optimal variable ordering σ^*, run it through InsideOut, for a total runtime of $\tilde{O}(N^{\mathsf{faqw}(\varphi)} + |\mathsf{output}|)$.

However, in some FAQ-framework's applications such as in graphical models, we cannot simply sweep query complexity under the rug. Moreover, computing the faqw is **NP**-hard because faqw is a strict generalization of the fractional hypertree width (fhtw), which is **NP**-hard [19] to compute. Hence, we find a good approximation for the faqw. This is the green path in Fig. 2: from the expression tree, we construct a tree decomposition for $\mathcal{H}$; then, from the GYO-elimination order of this tree decomposition we obtain a variable ordering $\bar{\sigma}$ for which we can show that $\mathsf{faqw}(\bar{\sigma}) \le \mathsf{faqw}(\sigma^*) + g(\mathsf{faqw}(\sigma^*))$, where g is any known approximation of fhtw (the best of which is due to Marx [26]).

Example 10. The variable ordering used earlier in Example 6 is a GYO-elimination order for the tree decomposition in Fig. 1b. (In GYO, when we eliminate variable X_n, the set U_n becomes a bag of the tree decomposition whose children are the bags corresponding to $\partial(n)$.) In particular, bags $\{f,h\}$, $\{e,g\}$, and $\{d,e,f\}$ resulted from eliminating h, g, and f respectively. The tree decomposition has width $\mathsf{fhtw} = 3/2$, same as the faqw of the variable ordering.

From these ideas, we obtained many corollaries, some of which are summarized in Table 1. The results in Table 1 span three areas: (1) CSPs and Logic; (2) PGMs and (3) Matrix operations. Except for joins, problems in area (1) need the full generality of FAQ formulation, where InsideOut either improves upon existing results or yields new results. Problems in area (2) can already be reduced to FAQ-SS. Here, InsideOut improves upon known results since it takes advantage of Grohe and Marx's more recent fractional hypertree width bounds. Finally, problems in area (3) of Table 1 are classic. InsideOut does not yield anything new here, but it is intriguing to be able to explain the textbook dynamic programming algorithm for Matrix-Chain Multiplication [15] as an algorithm to find a good variable ordering for the corresponding FAQ-instance. The DFT result is a re-writing of Aji and McEliece's observation [6].

4. PRACTICAL IMPLICATIONS

In this section we address two questions the readers might have regarding InsideOut: (1) hurdles one might face in a practical implementation of InsideOut, and (2) whether practical implications are as good as what the theory says.

Additional hurdles and how to solve them. There are a couple of problems we have to solve to implement InsideOut effectively.

The first problem is, in real-world queries, we do not just have materialized predicates as inputs, we also have predicates such as $a < b$, $a + b = c$, negations and so on. These predicates do not have a "size." To solve this problem, one solution is to set the "size" of those predicates to be ∞ while computing the AGM-bound. For instance, if we have a subquery of the form $Q \leftarrow R(a,b), S(b,c), a+b=c$, where R and S are input materialized predicates of size N, then by setting the size of $a+b=c$ to be infinite, $\mathsf{AGM}(Q) = N^2$. This solution does not work for two reasons. (1) If we knew $a+b=c$, then it is easy to infer that $|Q| \le N$ and also to compute Q in time $\tilde{O}(N)$: scan over tuples in R, use $a+b=c$ to compute c, and see if $(b,c) \in S$. In other words, the AGM-bound is no longer tight. (2) This solution may give an ∞-bound when the output size is clearly bounded. Consider, for example, the query $Q \leftarrow R(a), S(b), a+b=c$; in this case, $\{a,b,c\}$ is the only hyperedge covering vertex c in the fractional edge cover. Our implementation at LogicBlox

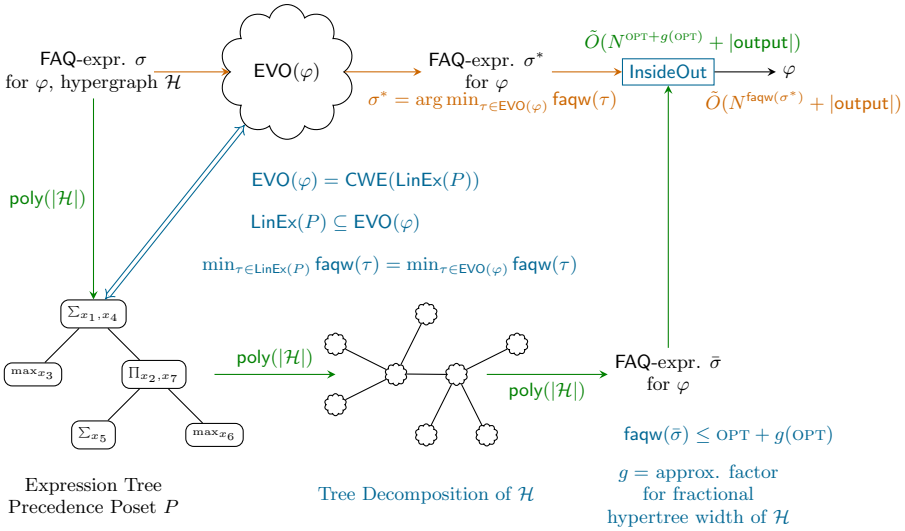


Figure 2: Sketch of main technical contributions

Problem	FAQ formulation	Previous Algo.	**Our Algo.**
#QCQ	$\sum_{(x_1,\ldots,x_f)} \bigoplus_{x_{f+1}}^{(f+1)} \cdots \bigoplus_{x_n}^{(n)} \prod_{S\in\mathcal{E}} \psi_S(\mathbf{x}_S)$ where $\bigoplus^{(i)} \in \{\max, \times\}$	No non-trivial algo	$\tilde{O}(N^{\mathsf{faqw}(\varphi)} + Z)$
QCQ	$\bigoplus_{x_{f+1}}^{(f+1)} \cdots \bigoplus_{x_n}^{(n)} \prod_{S\in\mathcal{E}} \psi_S(\mathbf{x}_S)$ where $\bigoplus^{(i)} \in \{\max, \times\}$	$\tilde{O}(N^{\mathsf{PW}(\mathcal{H})} + Z)$ [11]	$\tilde{O}(N^{\mathsf{faqw}(\varphi)} + Z)$
#CQ	$\sum_{(x_1,\ldots,x_f)} \max_{x_{f+1}} \cdots \max_{x_n} \prod_{S\in\mathcal{E}} \psi_S(\mathbf{x}_S)$	$\tilde{O}(N^{\mathsf{DM}(\mathcal{H})} + Z)$ [18]	$\tilde{O}(N^{\mathsf{faqw}(\varphi)} + Z)$
Joins	$\bigcup_{\mathbf{x}} \bigcap_{S\in\mathcal{E}} \psi_S(\mathbf{x}_S)$	$\tilde{O}\left(N^{\mathsf{fhtw}(\mathcal{H})} + Z\right)$ [21]	$\tilde{O}\left(N^{\mathsf{fhtw}(\mathcal{H})} + Z\right)$
Marginal Distribution	$\sum_{(x_{f+1},\ldots,x_n)} \prod_{S\in\mathcal{E}} \psi_S(\mathbf{x}_S)$	$\tilde{O}(N^{\mathsf{htw}(\varphi)} + Z)$ [23]	$\tilde{O}(N^{\mathsf{faqw}(\varphi)} + Z)$
MAP query	$\max_{(x_{f+1},\ldots,x_n)} \prod_{S\in\mathcal{E}} \psi_S(\mathbf{x}_S)$	$\tilde{O}(N^{\mathsf{htw}(\varphi)} + Z)$ [23]	$\tilde{O}(N^{\mathsf{faqw}(\varphi)} + Z)$
Matrix Chain Mult.	$\sum_{x_2,\ldots,x_n} \prod_{i=1}^{n} \psi_{i,i+1}(x_i, x_{i+1})$	DP bound [15]	DP bound
DFT	$\sum_{(y_0,\ldots,y_{m-1})\in\mathbb{Z}_p^m} b_y \cdot \prod_{0\le j+k<m} e^{i2\pi \frac{x_j \cdot y_k}{p^{m-j-k}}}$	$O(N \log_p N)$ [14]	$O(N \log_p N)$

Table 1: Runtimes of algorithms assuming optimal variable ordering is given. Problems shaded red are in CSPs and logic ($\mathbf{D} = \{0,1\}$ for CSP and $\mathbf{D} = \mathbb{N}$ for #CSP), problems shaded green fall under PGMs ($\mathbf{D} = \mathbb{R}_+$), and problems shaded blue fall under matrix operations ($\mathbf{D} = \mathbb{C}$). N denotes the size of the largest factor (assuming they are represented with the listing format). $\mathsf{htw}(\varphi)$ is the notion of integral cover width for PGM. $\mathsf{PW}(\mathcal{H})$ is the *optimal width of a prefix graph* of $\mathcal{H}$ and $\mathsf{DM}(\mathcal{H}) = \mathrm{poly}(F\text{-}\mathsf{ss}(\mathcal{H}), \mathsf{fhtw}(\mathcal{H}))$, where $F\text{-}\mathsf{ss}(\mathcal{H})$ is the $[f]$-quantified star size. Z is the output size in listing representation. Our width $\mathsf{faqw}(\varphi)$ is never worse than any of the three and there are classes of queries where ours is unboundedly better than all three. In DFT, $N = p^m$ is the length of the input vector. $\tilde{O}$ hides a factor of $\mathrm{poly}(|\mathcal{H}|) \cdot \log N$.

makes use of generalizations of AGM to queries with functional dependencies and immaterialized predicates (such as $a+b=c$). These new bounds are based on a linear program whose variables are marginal entropies [4, 5].

The second problem is to select a good variable ordering to run InsideOut on. In principle, one does not have to use the AGM-bound or the bounds from [4, 5] to estimate the cost of an FAQ subquery. If one were to implement InsideOut inside any RDBMS, one could poll that RDBMS's optimizer to figure out the cost of a given variable ordering. However, there are $n!$ variable orderings, and optimizer's cost estimation is time-consuming. Furthermore, some subqueries have inputs which are intermediate results. Hence, it is much faster to compute a variable ordering minimizing the faqw of the query, defined on the bounds in [4, 5]. As the problem is **NP**-hard, either an approximation algorithm [3] or a greedy heuristic suffices in our experience.

InsideOut is bottom-up dynamic programming. We can also solve FAQ queries with top-down (memoized) dynamic programming. In hindsight, this was the approach that Bak-

ibayev et al. [9] and Olteanu and Závodný [32] took to solve FAQ over a single semiring. We can also limit the amount of memoization in a top-down strategy to attain performance gain in some cases [22].

Practical Impact. It is trivial to construct classes of queries on real datasets for which InsideOut-style of algorithms gives arbitrarily large speedups over traditional RDBMSs. In fact, even when dynamic programming does not take effect, the speedup of backtracking search (and thus worst-case optimal algorithms) over traditional query plans is already huge [30]. The impact of the FAQ-framework and the InsideOut algorithm, however, go much beyond these toy queries (even when run on real datasets).

InsideOut is a component of LogicBlox's effort to extend LogiQL to be a probabilistic programming language [10], as part of DARPA's PPAML and MUSE programs. The component the algorithm handles is inference in discrete graphical models.

Learning from the beautiful work of Olteanu and Schleich [31, 34], we realized [27] that InsideOut can be used to train a large class of machine learning models *inside* the database. Our implementation showed orders of magnitude speedup over the traditional data modeler route of exporting the data and running it through `R` or `Python`. These models are trained with different variations of gradient descents, whose (pre-)computation steps are FAQ queries. What is much more interesting than the vanilla FAQ framework we presented above is that, in these applications, we want to compute *many* (in the 100K-range) FAQ queries all at once, making dynamic programming much more crucial to the performance. Another related approach was considered in [25].

5. CONCLUDING REMARKS

The FAQ framework showed that many common computational tasks over a very wide range of domains such as CSPs, machine learning, relational database, logic, and matrix computations can be performed *inside* a database using the same abstraction. The main idea is to blur the line between data and computation, as we use the database to store, compute, and process functions. The glue of the framework is a simple dynamic programming algorithm called InsideOut, which can be cast as a query-rewriting method and thus it is readily implementable within any RDBMS. These ideas are implemented, tested, and validated within the LogicBlox database system. Our theory predicts practice very well, which is a beautiful thing to see.

6. REFERENCES

[1] M. Abo Khamis, H. Q. Ngo, C. Ré, and A. Rudra. Joins via geometric resolutions: Worst-case and beyond. In *PODS*, pages 213–228. ACM, 2015.

[2] M. Abo Khamis, H. Q. Ngo, and A. Rudra. FAQ: questions asked frequently. *CoRR*, abs/1504.04044, 2015.

[3] M. Abo Khamis, H. Q. Ngo, and A. Rudra. FAQ: questions asked frequently. In *PODS*, 2016.

[4] M. Abo Khamis, H. Q. Ngo, and D. Suciu. Computing join queries with functional dependencies. In *PODS*, pages 327–342, 2016.

[5] M. Abo Khamis, H. Q. Ngo, and D. Suciu. What do shannon-type inequalities, submodular width, and disjunctive datalog have to do with one another? In *PODS*, 2017.

[6] S. M. Aji and R. J. McEliece. The generalized distributive law. *IEEE Transactions on Information Theory*, 46(2):325–343, 2000.

[7] M. Aref, B. ten Cate, T. J. Green, B. Kimelfeld, D. Olteanu, E. Pasalic, T. L. Veldhuizen, and G. Washburn. Design and implementation of the LogicBlox system. In *SIGMOD*, 2015.

[8] A. Atserias, M. Grohe, and D. Marx. Size bounds and query plans for relational joins. In *FOCS*, pages 739–748. IEEE Computer Society, 2008.

[9] N. Bakibayev, T. Kociský, D. Olteanu, and J. Zavodny. Aggregation and ordering in factorised databases. *PVLDB*, 6(14):1990–2001, 2013.

[10] V. Bárány, B. ten Cate, B. Kimelfeld, D. Olteanu, and Z. Vagena. Declarative probabilistic programming with datalog. In *ICDT*, pages 7:1–7:19, 2016.

[11] H. Chen and V. Dalmau. Decomposing quantified conjunctive (or disjunctive) formulas. In *LICS*, 2012.

[12] S. Chu, M. Balazinska, and D. Suciu. From theory to practice: Efficient join query evaluation in a parallel database system. In *SIGMOD*, pages 63–78, 2015.

[13] S. Cohen. User-defined aggregate functions: Bridging theory and practice. In *SIGMOD '06*, pages 49–60.

[14] J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, 19:297–301, 1965.

[15] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT Press, 2nd edition, 2001.

[16] M. Davis, G. Logemann, and D. W. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.

[17] R. Dechter. Bucket elimination: A unifying framework for reasoning. *Artif. Intell.*, 113(1-2):41–85, 1999.

[18] A. Durand and S. Mengel. Structural tractability of counting of solutions to conjunctive queries. In *ICDT*, pages 81–92, 2013.

[19] W. Fischl, G. Gottlob, and R. Pichler. General and Fractional Hypertree Decompositions: Hard and Easy Cases. *ArXiv e-prints*, Nov. 2016.

[20] S. W. Golomb and L. D. Baumert. Backtrack programming. *J. ACM*, 12(4):516–524, 1965.

[21] M. Grohe and D. Marx. Constraint solving via fractional edge covers. In *SODA*, pages 289–298, 2006.

[22] O. Kalinsky, Y. Etsion, and B. Kimelfeld. Flexible caching in trie joins, 2017. To appear in EDBT.

[23] K. Kask, R. Dechter, J. Larrosa, and A. Dechter. Unifying tree decompositions for reasoning in graphical models. *Artif. Intell.*, 166(1-2), 2005.

[24] D. Koller and N. Friedman. *Probabilistic graphical models*. Adaptive Computation and Machine Learning. MIT Press, 2009. Principles and techniques.

[25] A. Kumar, J. Naughton, and J. M. Patel. Learning generalized linear models over normalized data. In *SIGMOD*, pages 1969–1984. ACM, 2015.

[26] D. Marx. Approximating fractional hypertree width. *ACM Trans. Algorithms*, 6(2):29:1–29:17, Apr. 2010.

[27] H. Q. Ngo, X. Nguyen, D. Olteanu, and M. Schleich. In-database learning with sparse tensors, 2017. Manuscript.

[28] H. Q. Ngo, E. Porat, C. Ré, and A. Rudra. Worst-case optimal join algorithms. In *PODS*, pages 37–48, 2012.

[29] H. Q. Ngo, C. Ré, and A. Rudra. Skew strikes back: New developments in the theory of join algorithms. In *SIGMOD RECORD*, pages 5–16, 2013.

[30] D. T. Nguyen, M. Aref, M. Bravenboer, G. Kollias, H. Q. Ngo, C. Ré, and A. Rudra. Join processing for graph patterns: An old dog with new tricks. In *GRADES*, pages 2:1–2:8, 2015.

[31] D. Olteanu and M. Schleich. F: regression models over factorized views. *PVLDB*, 9(13):1573–1576, 2016.

[32] D. Olteanu and J. Závodný. Size bounds for factorised representations of query results. *ACM Trans. Datab. Syst.*, 40(1), 2015.

[33] F. Rossi, P. v. Beek, and T. Walsh. *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. Elsevier Science Inc., 2006.

[34] M. Schleich, D. Olteanu, and R. Ciucanu. Learning linear regression models over factorized joins. In *SIGMOD*, pages 3–18, 2016.

[35] T. L. Veldhuizen. Triejoin: A simple, worst-case optimal join algorithm. In *ICDT*, pages 96–106, 2014.

[36] N. Zhang and D. Poole. A simple approach to Bayesian network computations. In *Canadian AI*, pages 171–178, 1994.

[37] N. L. Zhang and D. Poole. Exploiting causal independence in Bayesian network inference. *J. Artificial Intelligence Res.*, 5:301–328, 1996.

Technical Perspective: Optimizing Tree Patterns for Querying Graph- and Tree-Structured Data

Benny Kimelfeld
Technion – Israel Institute of Technology
bennyk@cs.technion.ac.il

From the early days of databases, practitioners and researchers have pursued techniques for rewriting queries into equivalent ones that are easier to evaluate. The following paper closes a fundamental gap that we have had in our understanding of this challenge in the context of *tree patterns*. Such patterns are common and basic components of query languages for graph and tree data such as SPARQL, Cypher and XQuery. The authors study the question of whether the given tree pattern can be replaced with a smaller one, the question of whether it involves redundant conditions, and most importantly, the relationship between these two questions.

Formally, a tree pattern p is *matched* in a labeled graph G if the nodes of p can be mapped to the nodes of G in a way that all the *constraints* of p are satisfied. A *node constraint* is either a label match (e.g., the label is "person") or *wildcard* (no constraint), and an *edge constraint* is either *child* (the edge is mapped to an edge) or *descendant* (the edge is mapped to a path). Two patterns are *equivalent* if one is matched in a given graph precisely when the other does. The properties in focus are *minimality*—does p have the minimal size among all equivalent patterns? and *redundancy*—does the removal of any node of p (along with the subtree underneath) result in an equivalent pattern?

Tree-pattern minimization was studied by Flesca et al. [2] in 2003, where it was claimed that minimization can be achieved through containment tests among sub-patterns. Moreover, their results imply that determining whether a tree pattern is minimal is an NP-complete problem. It was not before 2008 that Kimelfeld and Sagiv [4] established that Flesca et al. [2] had a gap in their arguments, as their results apply to *nonredundancy* and not *minimality*, and in fact, the case of minimality was still open.

Nevertheless, Kimelfeld and Sagiv believed that the results of Flesca et al. [2] were valid, and formulated a conjecture that the following paper refers to as the *M-NR conjecture*: minimality (M) and nonredundancy (NR) are the same, that is, a tree pattern is minimal if and only if it is nonredundant [4]. The conjecture was supported by various classes of tree patterns where it was proved to hold true [3,4]. Yet, correctness of the conjecture in general remained open. Verifying the conjecture required showing that every nonredundant pattern is minimal (as the other direction is clearly true). Refuting the conjecture required finding a single counterexample. To our surprise, one was indeed found.

The following paper highlights a publication in 2016 ACM PODS, where Czerwinski et al. [1] refuted the M-NR conjecture. Notwithstanding the time it took to find it, their counterexample is fairly simple and small (enough to fit in T-shirts worn by the authors of [1] during the conference); it is a nonredundant tree pattern of 32 nodes, and they show an equivalent one with only 31 nodes. As they further show, not only are the two properties different, the corresponding computational decision problems are fundamentally different (under conventional complexity assumptions), again in contrast to past beliefs [3, 4]: while nonredundancy is NP-complete, minimality is Π_2^{p}-complete (hence, resides higher in the polynomial hierarchy). In particular, it follows that one needs tools beyond containment tests if minimization of general tree patterns is desired. That and more in what follows.

1. REFERENCES

[1] W. Czerwinski, W. Martens, M. Niewerth, and P. Parys. Minimization of tree pattern queries. In *PODS*, pages 43–54. ACM, 2016.
[2] S. Flesca, F. Furfaro, and E. Masciari. On the minimization of XPath queries. In *VLDB*, pages 153–164, 2003.
[3] S. Flesca, F. Furfaro, and E. Masciari. On the minimization of XPath queries. *J. ACM*, 55(1):2:1–2:46, 2008.
[4] B. Kimelfeld and Y. Sagiv. Revisiting redundancy and minimization in an XPath fragment. In *EDBT*, pages 61–72. ACM, 2008.

Optimizing Tree Patterns for Querying Graph- and Tree-Structured Data*

Wojciech Czerwiński†
University of Warsaw
wczerwin@mimuw.edu.pl

Wim Martens
Universität Bayreuth
wim.martens@uni-bayreuth.de

Matthias Niewerth‡
Universität Bayreuth
matthias.niewerth@uni-bayreuth.de

Paweł Parys
University of Warsaw
parys@mimuw.edu.pl

ABSTRACT

Many of today's graph query languages are based on graph pattern matching. We investigate optimization for tree-shaped patterns with transitive closure. Such patterns are quite expressive, yet can be evaluated efficiently. The *minimization* problem aims at reducing the number of nodes in patterns and goes back to the early 2000's. We provide an example showing that, in contrast to earlier claims, tree patterns cannot be minimized by deleting nodes only. The example resolves the M $\stackrel{?}{=}$ NR problem, which asks if a tree pattern is minimal if and only if it is nonredundant. The example can be adapted to also understand the complexity of minimization, which was another question that was open since the early research on the problem. Interestingly, the latter result also shows that, unless standard complexity assumptions are false, more general approaches for minimizing tree patterns are also bound to fail in some cases.

1. INTRODUCTION

Tree patterns are a very natural and user-friendly means to query graph- and tree-structured data. This is why they can be found in the conceptual core of widely used query languages for graphs and trees.

1.1 Motivation from Graph Query Languages

Graph pattern matching is a fundamental concept in modern declarative graph query languages. Indeed, graph query languages usually take one of two main perspectives: *graph traversal* or *graph pattern matching*, the former being the imperative and the latter being the declarative variant [31]. Today's most prominent declarative graph query languages are SPARQL 1.1 [33] and Neo4J Cypher [25]. Both languages make it very clear in their specifications that they have graph pattern matching at their core. SPARQL 1.1 explicitly writes "SPARQL is based around graph pattern matching" [33, Section 5], and the introduction of Neo4J's documentation on Cypher [25, Section 3.1.1] is essentially an introduction to the principles of graph pattern matching. Gremlin [19], another popular graph query language, leans more towards the graph traversal side of the spectrum, but also supports pattern matching style querying. It performs graph pattern matching similar to SPARQL [31].

The reason why graph pattern matching is so popular is not surprising. Graph patterns are expressive, reasonably simple and intuitive to understand, and often efficient to evaluate. Consider the graph in Figure 1. It contains information on artists, their occupation, and their place of birth. The graph structure is inspired on *property graphs*, a popular model for graph databases in practice [30, 3]. In this model, each node and edge carry a label and, in addition, nodes can have a set of attributes. For instance, the node related to Jimi Hendrix has the label Person, its "name" attribute is Jimi Hendrix, and its "aka" attribute is James Marshall Hendrix.

Assume that we would like to find the artists who were born in the United States. This corresponds to finding names of Person nodes that have (1) an occupation edge to "a subclass of artist" and (2) a place of birth edge to a city that is located in the United States. For expressing these conditions, we need to reason about paths in the graph. The occupation in (1) should be connected to artist by a path of subclassof-edges and the city in (2) to United States by a path of locatedin-edges.

These conditions are expressed in the pattern in Figure 2.[1] It has two types of edges and two types of nodes. Single

*The original version of this article was published in PODS 2016, titled "Minimization of tree pattern queries" [12].

†Supported by Poland's National Science Centre grant no. UMO-2013/11/D/ST6/03075.

‡Supported by grant number MA 4938/2–1 from the Deutsche Forschungsgemeinschaft (Emmy Noether Nachwuchsgruppe).

[1]The pattern is closely related to *graph patterns*, which were identified by Angles et al. [3] as a part of the conceptual core of many of today's graph query languages.

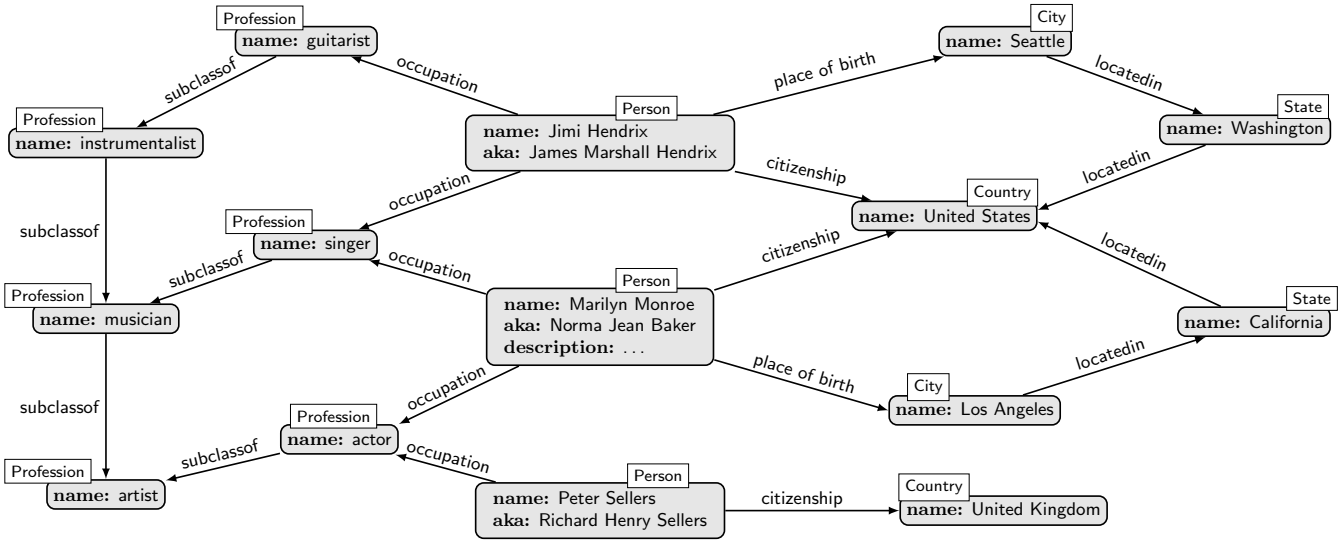


Figure 1: A graph database (as a *property graph*), inspired on a fragment of WikiData

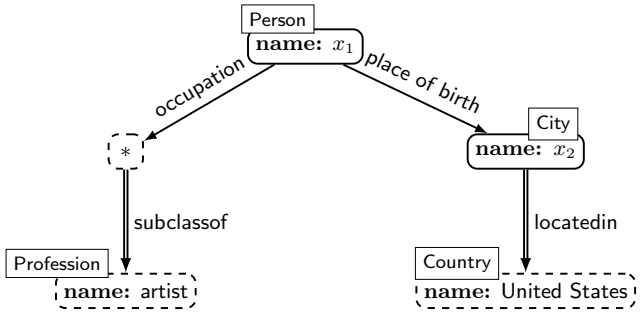


Figure 2: A tree pattern finding the artists who were born in the United States. The query returns the person names and the cities where they were born. (Fully circled nodes are return nodes.)

edges in the pattern can be matched to single edges in the graph with the same label. The double edges can be matched to *paths* in the graph on which every edge has the label given in the query. (For instance, the locatedin edge in the query can be matched on the path from the Los Angeles to United States nodes.) The solid nodes in the query are *output nodes* and the dashed nodes are ordinary nodes. The symbol $*$ is a wildcard symbol that can be matched to any label. The query has two variables: x_1 and x_2. Intuitively, computing the answers to the pattern corresponds to finding *matches* of the pattern in the graph and, for each such match, return the nodes (or values) matched by the variables in output nodes of the pattern. When evaluated on the graph in Figure 1, this pattern would return (Jimi Hendrix, Seattle) and (Marilyn Monroe, Los Angeles).

Our example query is structured as a tree. In general, the underlying structure of queries in SPARQL or Cypher can be an arbitrary graph and can therefore contain cycles. The acyclic queries form, however, an important subclass. Graph patterns closely correspond to *conjunctive queries*, which are known to be NP-complete to evaluate [10]. The tree-shaped patterns closely correspond to *acyclic conjunctive queries*, which can be evaluated in polynomial time. In fact, the quest for subclasses of conjunctive queries with a polynomial time evaluation problem is rich of beautiful results (see, e.g., [17]). In this paper, however, we focus on queries whose underlying structure is a tree and, for this reason, have a tractable (polynomial time) evaluation problem. (We note that the transitive closure operators we use make no difference in this respect.)

From a graph query language perspective, the tree patterns from this paper correspond to tree-shaped conjunctive queries (or tree-shaped graph patterns) with transitive closure. Transitive closure seems to be becoming increasingly popular in graph query languages, even though there have been challenges in the early version of the operator in SPARQL 1.1 [5, 23]. In WikiData's list of *example queries* [34], which help users getting started with the data set, 72 out of 272 queries use transitive closure of a label, which means that the feature is important.

1.2 Motivation from Tree Query Languages

Tree-structured data is among us in many forms, JSON and XML being two examples. The tree pattern queries that we consider were originally introduced to investigate query languages for tree-structured data [24]. They are an abstraction of a fragment of XPath [28] and therefore also appear in XQuery [29], XSLT [21], and languages for querying JSON, see, e.g., [20]. Indeed, patterns such as the one in Figure 2 can equally well be used for querying tree-structured data. (This is easy to see, since a tree is a special case of a graph.)

Tree pattern queries are also important for many topics in fundamental research on tree-structured data. For instance, they form a basis for conjunctive queries over trees [18, 8], for models of XML with incomplete information [6], and the closely related pattern-based XML queries [16]. They are used for specifying guards in Active XML systems [1] and for specifying schema mappings in XML data exchange [4].

1.3 The Core Problem

We report in this paper on recent progress on the minimization problem for tree patterns [12]. Optimization of queries has been a main topic of database research ever since

the beginning and therefore is very natural to consider for tree patterns. Tree pattern query optimization already attracted significant attention in the form of *query containment* [24, 26, 13], *satisfiability* [7], and *minimization* [2, 11, 15, 22, 27, 35].

Almost all this former work on containment, satisfiability, and minimization exclusively considered tree patterns as a language for querying *tree-structured data*. However, as argued by Miklau and Suciu [24, Section 5.3], many of these results hold just the same if we use tree patterns to query *graph-structured data*, i.e., if we use tree patterns as in Section 1.1. The same argument holds for the minimization problem. For this reason, one can often obtain results for tree patterns on graph-structured data while only considering tree-structured data in proofs.

We note that the tree patterns that were considered in this former work (and the ones we consider in the proofs of [12]) cannot express the query in Figure 2, for the simple reason that they cannot express the transitive closure of *subclassof*. We will argue that our results extend to these more expressive queries as well.

Another difference is that we consider Boolean queries, whereas the query in Figure 2 returns tuples of answers. Again, we will argue that our results also apply for higher-arity queries. We consider the following problem.

Tree Pattern Minimization	
Given:	A tree pattern p and $k \in \mathbb{N}$
Question:	Is there a tree pattern q, equivalent to p, such that its size is at most k?

The main difficulties for this problem are already present in a very restricted set of tree patterns that

- only query *graphs that are node-labeled and are tree-shaped*; and
- over these graphs, only use *labeled node tests*, *wildcard node tests*, the *child relation*, and the *descendant relation*.

These are precisely the patterns introduced by Miklau and Suciu [24].

1.4 History of the Problem

Although the patterns we consider here have been widely studied [14, 24, 36, 15, 22, 1, 9, 4, 32], their minimization problem remained elusive for a long time. The most important previous work for their minimization was done by Kimelfeld and Sagiv [22] and by Flesca, Furfaro, and Masciari [14, 15].

The key challenge was understanding the relationship between *minimality* (M) and *nonredundancy* (NR). Here, a tree pattern is minimal if it has the smallest number of nodes among all equivalent tree patterns. It is nonredundant if none of its leaves (or branches[2]) can be deleted while remaining equivalent. The question was if minimality and nonredundancy are the same ([22, Section 7] and [15, p. 35]):

M $\stackrel{?}{=}$ NR Problem:
Is a tree pattern minimal if and only if it is nonredundant?

[2]Kimelfeld and Sagiv proved that a tree pattern has a redundant branch if and only if it has a redundant leaf [22, Proposition 3.3].

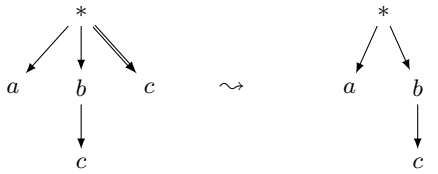


Figure 3: Minimizing a tree pattern by removing redundant nodes

Notice that a part of the M $\stackrel{?}{=}$ NR problem is easy to see: a minimal pattern is trivially also nonredundant (that is, M $\subseteq$ NR). The opposite direction is much less clear.

If the problem would have a positive answer, it would mean that the simple algorithmic idea summarised in Algorithm 1 correctly minimizes tree patterns. Therefore, the M $\stackrel{?}{=}$ NR problem is a natural question about the design of minimization algorithms for tree patterns.

Algorithm 1 Computing a nonredundant subpattern

Input: A tree pattern p
Output: A nonredundant tree pattern q, equivalent to p

while a leaf of p can be removed (remaining equivalent to p) **do**
  Remove the leaf
end while
return the resulting pattern

Example 1.1. *It is easy to see that Algorithm 1 can be used for minimizing some patterns. Consider the left pattern in Figure 3. Its root (labeled with a wildcard $*$) can be matched on nodes n in a graph such that (1) n has an a-labeled successor, (2) a b-labeled successor with a c-labeled successor, and (3) a c-labeled node is reachable from n. (In this example, edge labels do not matter.) In the semantics of such patterns, it is allowed that the different c-nodes are matched on the same node in the data. Therefore, condition (3) is redundant and the pattern to the right is equivalent and smaller.*

The M $\stackrel{?}{=}$ NR problem is also a question about complexity. The main source of complexity of the nonredundancy algorithm lies in testing equivalence between a pattern p and a pattern p', which is generally coNP-complete [24]. If M $\stackrel{?}{=}$ NR has a positive answer, then Tree Pattern Minimization would also be coNP-complete.

In fact, the problem was claimed to be coNP-complete in 2003 [14, Theorem 2], but the status of the minimization- and the M $\stackrel{?}{=}$ NR problems were re-opened by Kimelfeld and Sagiv [22], who found errors in the proofs. Flesca et al.'s journal paper then proved that M = NR for a limited class of tree patterns, namely those where *every wildcard node has at most one child* [15]. Nevertheless, for tree patterns,

(a) the status of the M $\stackrel{?}{=}$ NR problem and

(b) the complexity of the minimization problem

remained open.

1.5 Our Contributions

We proved the following [12]:

(a) There exists a tree pattern that is nonredundant but not minimal. Therefore, M $\neq$ NR.

(b) Tree Pattern Minimization is Σ_2^P-complete. This implies that even the main idea in Algorithm 1 cannot work unless coNP $= \Sigma_2^P$.

Interestingly, our counterexample for (a) uses only two wildcard nodes with two children and only one transitive edge. This is only barely beyond the fragment for which it is known that minimality and nonredundancy coincide.

Outline.

In Section 2 we formally define tree patterns, their semantics, and discuss their relationship to the queries in the Introduction. We show why M $\neq$ NR in Section 3. In Section 4 we briefly discuss the complexity result and its consequences.

2. PRELIMINARIES

We formally define our data model and queries, recall important results about the static analysis of queries, and discuss the relationship between other data models and ours.

Data Model: Node- and Edge-Labeled Graphs.

Our data model is very simple: we use finite, node- and edge-labeled directed graphs, where the labels come from an infinite set. In the graph database world, this model is closely related to *property graphs*, the data model for Neo4J [30] (see, e.g., [3] for a formal definition of property graphs).[3]

More formally, a *(node- and edge-) labeled graph* is a triple (V, E, lab), where V is a finite nonempty set of *nodes*, E is a set of directed *edges* $(u, v) \in V \times V$ and $\text{lab} : V \cup E \to \Lambda$ is a *labeling function* assigning to every node and edge its label coming from an infinite set of labels Λ. We assume that graphs are connected. A *path* from node v_1 to v_n is a sequence of nodes $\pi = v_1 \cdots v_n$, where $(v_i, v_{i+1}) \in E$ for every $i = 1, \ldots, n-1$.

A graph is a *tree* if,

(i) for every node v, there is at most one node u (called *parent* of v) with $(u, v) \in E$ and

(ii) there is exactly one node v (called *root*) without a parent.

We assume familiarity with standard terminology on trees such as *child* and *descendant*.

The Queries: Tree Patterns.

Our formal model of graph patterns allows node- and edge label tests, wildcard tests, and transitive closures. The wildcard test (denoted by "$*$" in patterns) matches any node- or edge label in a graph. To avoid confusion, we assume that $* \notin \Lambda$.

[3]Property graphs are more refined, however, since they associate *properties* to nodes in addition to labels. From a formal perspective, we want that nodes in the graph are not uniquely determined by their label. We do not want that different occurrences of a label in a query must always be mapped to the same node in the graph. This behaviour would introduce unwanted cycles in tree pattern queries.

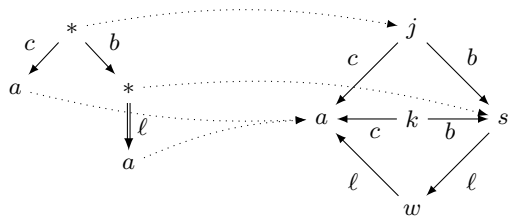


Figure 4: Example of a match from a tree pattern (left) to a labeled graph (right)

Formally, a *graph pattern* is a tuple $p = (V_p, E_p, \text{lab}_p)$ where $\text{lab}_p : V_p \cup E_p \to \Lambda \uplus \{*\}$ and V_p is partitioned in two sets: *simple edges* and *transitive closure edges*. In figures, we draw transitive closure edges using double lines. Furthermore, if we do not write a label on an edge, we implicitly assume that the edge label is the wildcard "$*$".

A *tree pattern* is a graph pattern that satisfies the conditions (i) and (ii) we required for trees. From now on in this paper, we will only consider *tree patterns* (although many definitions also apply for graph patterns). The *size* of a pattern p, denoted size(p), is the number of its nodes.

For simplicity, we will define our queries to be Boolean, that is, we will only consider whether they can be matched in a graph or not. Tree patterns with output nodes have been considered as well [24, 22] and our main results also apply to those queries. We discuss this later in the Preliminaries (see *Boolean vs. k-ary queries*).

Semantics of Queries.

We use a homomorphism-based semantics for tree patterns. For a tree pattern $p = (V_p, E_p, \text{lab}_p)$ and a graph $G = (V, E, \text{lab})$, a function $\text{m} : V_p \to V$ is a *match* of p in g if it fulfills all the following conditions:

(1) If $\text{lab}_p(v) \neq *$ for $v \in V_p$ then $\text{lab}_p(v) = \text{lab}(\text{m}(v))$.

(2) If $(u, v) \in E_p$ is a simple edge then $(\text{m}(u), \text{m}(v))$ is an edge in G. Furthermore, if $\text{lab}_p((u, v)) \neq *$ then $\text{lab}_p((u, v)) = \text{lab}((\text{m}(u), \text{m}(v)))$.

(3) If $(u, v) \in E_p$ is a transitive closure edge then there is a path from $\text{m}(u)$ to $\text{m}(v)$ in G that satisfies the label constraint of the edge. That is, there exists a path $\pi = u_1 \cdots u_n$ in G (with $n > 1$)) such that $\text{m}(u) = u_1$ and $\text{m}(v) = u_n$. Furthermore, if $\text{lab}_p((u, v)) \neq *$, then all edges (u_i, u_{i+1}) in π are labeled $\text{lab}_p((u, v))$.

We say that p *can be matched in* G if there exists a match from p to G. Figure 4 shows an example of a match. Notice that we do not require matches to be injective.

Definition 2.1 (Semantics of Tree Patterns). The set of *models of a tree pattern* p, denoted by $M(p)$, is the set of graphs in which p can be matched.

Containment, Equivalence, and Minimality.

A tree pattern p_1 is *contained* in a tree pattern p_2 if $M(p_1) \subseteq M(p_2)$, which we denote by $p_1 \subseteq p_2$. If $p_1 \subseteq p_2$ and $p_1 \supseteq p_2$ then we say that the patterns p_1 and p_2 are *equivalent* and we write $p_1 \equiv p_2$.

Figure 3 contains two patterns that are equivalent. (For the left pattern, the c-labeled node on the right branch can

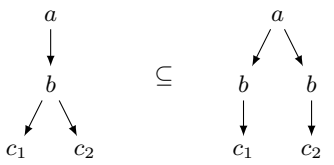


Figure 5: Example for containment of patterns. (Non-labeled edges are implicitly assumed to have wildcard tests.)

always be matched to wherever the c-labeled node in the middle branch is matched. Therefore it is equivalent to the pattern on the right.) In Figure 5, we give an example for pattern containment. The right pattern matches a-nodes which have c_1- and c_2-nodes on distance two, such that there are b-nodes between the a and the c_i. The pattern on the left additionally requires the two b-nodes to be the same. Since the latter is more restrictive, if the left pattern can be matched in a graph, then the right one can be matched there as well.

The following problem is important in many query optimization procedures:

Tree Pattern Equivalence	
Given:	Two tree patterns p_1 and p_2
Question:	Is $p_1 \equiv p_2$?

We call a tree pattern p *redundant* if one of its nodes can be removed without changing its set of models. For a node v of p, we denote by $p \setminus v$ the pattern obtained from p by removing v and all its descendants and incident edges.

Definition 2.2 (Minimality, Nonredundancy).

- A tree pattern p is *redundant* if it is equivalent to $p \setminus v$ for a node v of p. In this case, v is a *redundant node*. If p is not redundant we say that it is *nonredundant*.
- A pattern p is said to be *minimal* if there exists no tree pattern that is equivalent to p but has strictly smaller size.

It is known that tree patterns are redundant if and only if they have a redundant leaf [22, Proposition 3.3].

Complexity.

One can obtain an almost trivial Σ_2^P upper bound for Tree Pattern Minimization (as defined in the Introduction) by using the following result.

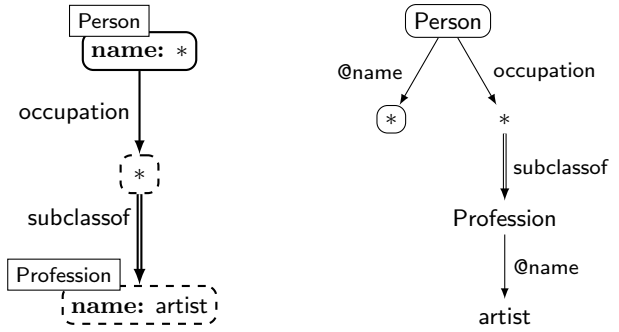
Theorem 2.3. Tree Pattern Equivalence *is* coNP*-complete.*

Proof sketch. Miklau and Suciu [24] prove this theorem for tree patterns without edge labels, but these can easily be added. Furthermore, their patterns only have tree models, whereas we consider graph models. However, they explain [24, Section 5.3] that these two variants of the problem are the same. □

From this result, a Σ_2^P upper bound for Tree Pattern Minimization is immediate.

Theorem 2.4. Tree Pattern Minimization *is in* Σ_2^P.

Proof. Given a tree pattern p and $k \in \mathbb{N}$, the Σ_2^P algorithm first guesses (existential quantification) a tree pattern p' of size at most k and then checks (universal quantification) if p' and p are equivalent. □



(a) Property Graph Query (b) Translated Query

Figure 6: Translating a subquery of Figure 2 to our simplified model

Notice that, if M = NR, then p' can be found among the subpatterns of p, which would drop the upper bound to coNP.

Boolean vs. k-ary queries.

One can easily extend tree patterns to k-ary tree patterns that return k-tuples of answers (see, e.g., [24, 22]). We argue that our results also hold for such queries. It is trivial for our M ≠ NR example, because a Boolean query is just a special case of a k-ary query. The other main result is the Σ_2^P-completeness result in Theorem 4.1. The Σ_2^P upper bound can be seen to hold for k-ary queries by using the same naive algorithm as in Theorem 2.4 and using the argument of Kimelfeld and Sagiv [22, Section 5.2] for showing that Tree Pattern Equivalence for k-ary queries polynomially reduces to the same problem for Boolean queries. The Σ_2^P lower bound follows immediately.

Relationship to the Queries in the Introduction.

The tree patterns we defined here are much simpler than the pattern we discussed in the Introduction (Figure 2). However, the two types of patterns are closely related when it comes to minimization. Again, since the patterns we have here are simpler, it is easy to see that our M ≠ NR example equally applies to the kind of patterns in the Introduction.

Moreover, the simplified patterns capture much of the expressivity of the more complex patterns modulo a simple encoding. In Figure 6, we demonstrate this translation by example, using a subquery of Figure 2. Essentially, each node of the pattern on the left becomes a node on the right labeled with the *property* (the label in the rectangular box) if present, and the "name"-attributes of nodes become children with incoming edges that identify the type of attribute. (We can make sure that the labels of these incoming edges do not appear elsewhere in the query.)

We do not claim that this translation gives a 100% correspondence between the world of tree patterns and the world of "property graph tree patterns", but we do believe that it shows a very close connection. For instance, the translation can be used for testing equivalence between certain types of property graph patterns (translate to tree patterns and test equivalence between those). Likewise, for a large class of property graph tree patterns, minimization would work very similarly to minimization of the translated tree pattern query.

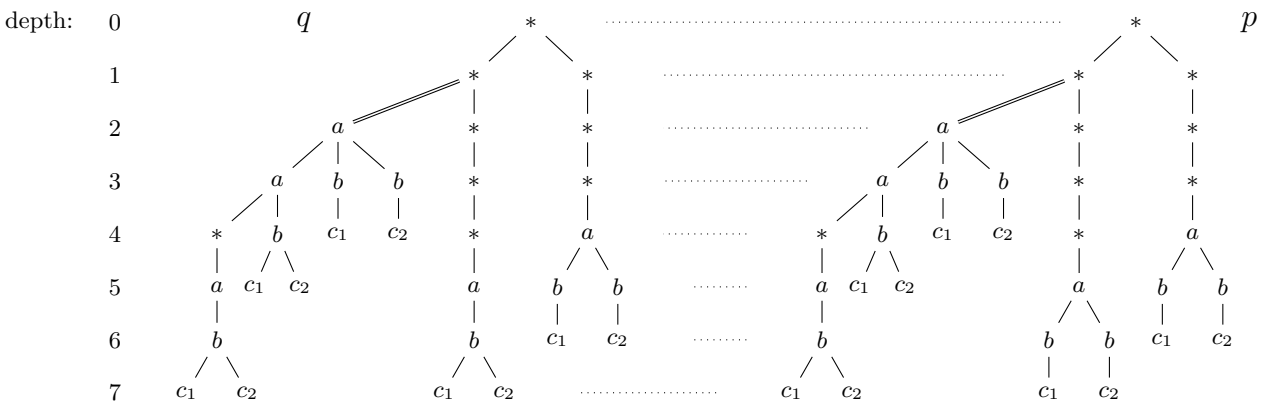


Figure 7: A non-redundant tree pattern p (right) and an equivalent tree pattern q that is smaller (left)

3. THE M $\stackrel{?}{=}$ NR PROBLEM

We show that M $\neq$ NR by presenting a tree pattern that is nonredundant but also not minimal.

Indeed, we will argue that the right pattern p in Figure 7 is nonredundant and not minimal. (For readability, we omitted arrows. All arrows are assumed to point downwards.) Consider the pattern q on the left of Figure 7. To convince the reader, we need to make three points: (1) p is nonredundant, (2) p is equivalent to q, and (3) q is smaller than p.

Point (3) is trivial: q can be obtained from p by merging two b-nodes on depth six. Therefore, q has one fewer node than p. Points (1) and (2) are non-trivial. Here we will only show (2) because it is the most interesting argument of the two. (Point (1) can be shown by proving that p is not equivalent to any of its subpatterns, see [12].)

We want to convince the reader of point (2) by a sequence of pictures. First of all, observe that $q \subseteq p$. The reason is the same as the one we already discussed in Figure 5. Therefore it only remains to argue why $p \subseteq q$.

In Figure 8, we depicted q (always on the left) and three patterns p_1, p_2 and p_3 on the right. If p is matched in a graph, there are three possibilities for matching the double edge connecting the $*$-node with the a-node. This double edge is matched to a path that either consists of

(a) one edge,

(b) two edges, or

(c) at least three edges.

These three possibilities are depicted on the right of Figure 8. If we have case (a), then we can also match the left pattern in Figure 8(a) (similar for (b) and (c)). (Some parts of these patterns are grey. We will get to that soon.)

The dotted edges have the following meaning. Whenever the pattern p_1, p_2, or p_3 on the right can be matched on a graph, then pattern q (on the left) can also be matched, by matching the nodes on the left to wherever the connected node on the right is matched. For instance, in case (a), the root of q can always be matched to wherever the root of p_1 was matched. The grey part of p_1 is in fact irrelevant for q in this case. All nodes of q can be matched to places where black nodes of p_1 are matched. The grey parts in (b) and (c) have the same meaning.

The dotted edges show completely how q can be matched in cases (a) and (b). In case (c), we also have a dashed edge. The dashed edge shows how the matching of q works if we have *exactly* three edges in (c), but if there are more, then the target of the edge needs to go downward accordingly. The reason for this is easy to see: the two a-nodes on the right side of q are connected to the root by paths of fixed length. So, if the target of the a-nodes move further away, the root of q needs to follow as well. Since all nodes on the path to the root are wildcards, this is possible. Therefore, q can always be matched in case (c) as well.

This gives us the following Theorem:

THEOREM 3.1 (M $\neq$ NR).
MINIMALITY $\neq$ NONREDUNDANCY

4. COMPLEXITY AND CONSEQUENCES

Leveraging the behavior of the patterns in Figure 7, we could prove the following:

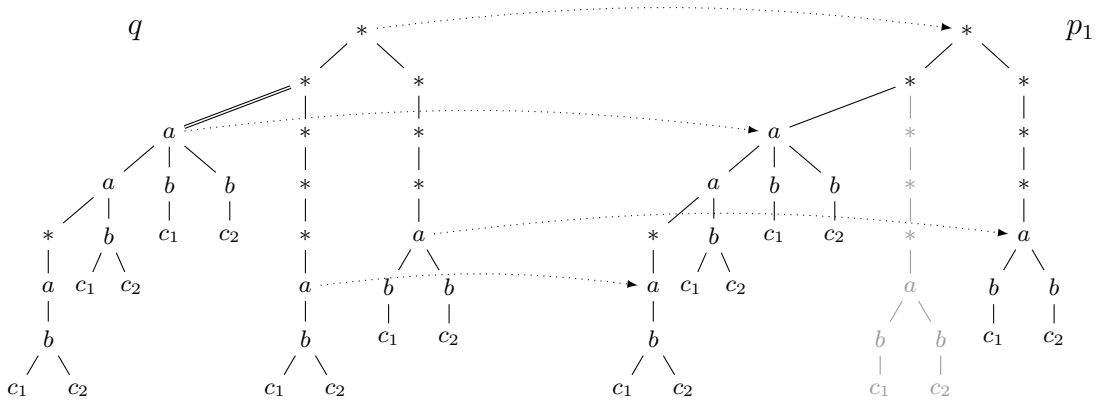
THEOREM 4.1 ([12]). TREE PATTERN MINIMIZATION *is* Σ_2^P*-complete.*

This result is even more drastic than the example in Figure 7. Observe that the query q can be obtained from p by just merging two nodes together. So, the reader may wonder if the following is true. Say that a query is in NR$'$ if none of its nodes can be *deleted or merged* while remaining equivalent. Then, M $\stackrel{?}{=}$ NR$'$ would be the question: *Can tree patterns always be minimized by deleting or merging nodes?*

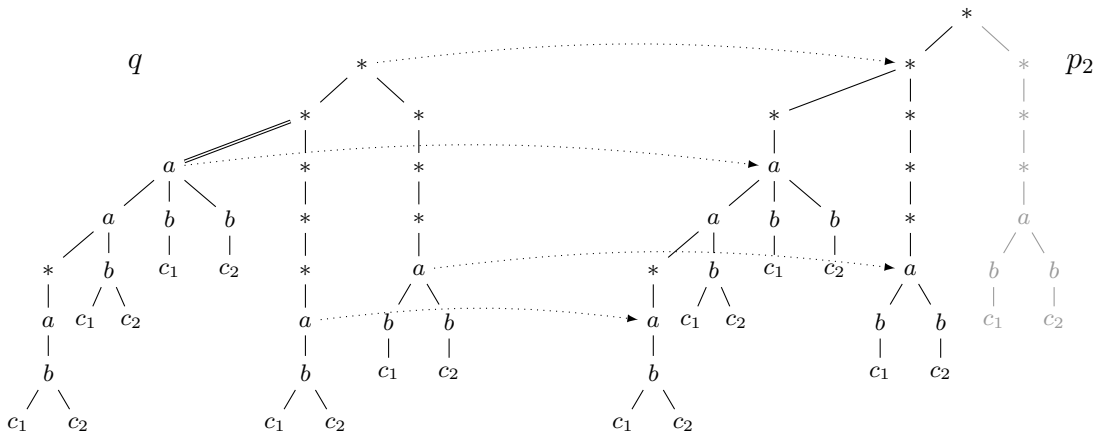
Although Figure 7 does not show that M $\neq$ NR$'$, Theorem 4.1 shows that, if M = NR$'$, then coNP = Σ_2^P. Indeed, if it *would* be possible to always minimize tree patterns by deleting or merging nodes, then Algorithm 1 (from the Introduction) can be adapted to be a coNP test for minimization. (Instead of deleting nodes, it would also merge nodes together.) For this reason, also the *search* for candidate minimal patterns is a difficult problem.

Acknowledgments

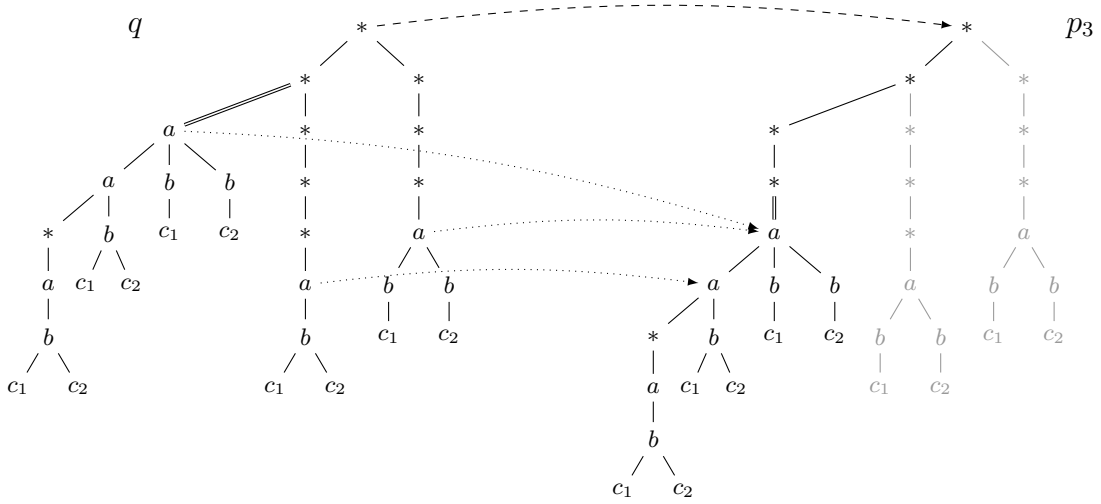
We are very grateful to Benny Kimelfeld for insightful discussions and for bringing the tree pattern minimization problem to our attention. We thank Dominik D. Freydenberger for carefully proofreading a draft of the paper.



(a) How q can be matched if p_1 can be matched



(b) How q can be matched if p_2 can be matched



(c) How q can be matched if p_3 can be matched

Figure 8: Showing that patterns p and q in Figure 7 are equivalent

5. REFERENCES

[1] S. Abiteboul, L. Segoufin, and V. Vianu. Static analysis of active XML systems. *ACM Trans. Database Syst.*, 34(4), 2009.

[2] S. Amer-Yahia, S. Cho, L. V. S. Lakshmanan, and D. Srivastava. Tree pattern query minimization. *VLDB J.*, 11(4):315–331, 2002.

[3] R. Angles, M. Arenas, P. Barceló, A. Hogan, J. L. Reutter, and D. Vrgoc. Foundations of modern graph query languages. *CoRR*, abs/1610.06264, 2016.

[4] M. Arenas, P. Barceló, L. Libkin, and F. Murlak. *Foundations of Data Exchange*. Cambridge University Press, 2014.

[5] 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 *World Wide Web Conference (WWW)*, pages 629–638, 2012.

[6] P. Barceló, L. Libkin, A. Poggi, and C. Sirangelo. XML with incomplete information. *J. ACM*, 58(1):4, 2010.

[7] M. Benedikt, W. Fan, and F. Geerts. XPath satisfiability in the presence of DTDs. *J. ACM*, 55(2), 2008.

[8] H. Björklund, W. Martens, and T. Schwentick. Conjunctive query containment over trees. *J. Comput. Syst. Sci.*, 77(3):450–472, 2011.

[9] H. Björklund, W. Martens, and T. Schwentick. Validity of tree pattern queries with respect to schema information. In *International Symposium on Mathematical Foundations of Computer Science (MFCS)*, pages 171–182, 2013.

[10] A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *Symposium on Theory of Computing (STOC)*, pages 77–90, 1977.

[11] D. Chen and C. Y. Chan. Minimization of tree pattern queries with constraints. In *International Conference on Management of Data (SIGMOD)*, pages 609–622, 2008.

[12] W. Czerwiński, W. Martens, M. Niewerth, and P. Parys. Minimization of tree pattern queries. In *Symposium on Principles of Database Systems (PODS)*, pages 43–54, 2016.

[13] W. Czerwiński, W. Martens, P. Parys, and M. Przybyłko. The (almost) complete guide to tree pattern containment. In *Symposium on Principles of Database Systems (PODS)*, pages 117–130, 2015.

[14] S. Flesca, F. Furfaro, and E. Masciari. On the minimization of XPath queries. In *International Conference on Very Large Data Bases (VLDB)*, pages 153–164, 2003.

[15] S. Flesca, F. Furfaro, and E. Masciari. On the minimization of XPath queries. *J. ACM*, 55(1), 2008.

[16] A. Gheerbrant, L. Libkin, and C. Sirangelo. Reasoning about pattern-based XML queries. In *International Conference on Web Reasoning and Rule Systems (RR)*, pages 4–18, 2013.

[17] G. Gottlob, G. Greco, N. Leone, and F. Scarcello. Hypertree decompositions: Questions and answers. In *Symposium on Principles of Database Systems (PODS)*, pages 57–74, 2016.

[18] G. Gottlob, C. Koch, and K. U. Schulz. Conjunctive queries over trees. *J. ACM*, 53(2):238–272, 2006.

[19] Gremlin Language. `github.com/tinkerpop/gremlin/wiki`, 2013.

[20] `jsonpath.com/`, January 2017.

[21] M. Kay. XSL Transformations (XSLT) version 3.0. Technical report, World Wide Web Consortium, November 2015. W3C Recommendation, `www.w3.org/TR/2015/CR-xslt-30-20151119/`.

[22] B. Kimelfeld and Y. Sagiv. Revisiting redundancy and minimization in an XPath fragment. In *International Conference on Extending Database Technology (EDBT)*, pages 61–72, 2008.

[23] K. Losemann and W. Martens. The complexity of regular expressions and property paths in SPARQL. *ACM Trans. Database Syst.*, 38(4):24, 2013.

[24] G. Miklau and D. Suciu. Containment and equivalence for a fragment of XPath. *J. ACM*, 51(1):2–45, 2004.

[25] The Cypher query language. `neo4j.com/docs/developer-manual/current/cypher/`, 2016.

[26] F. Neven and T. Schwentick. On the complexity of XPath containment in the presence of disjunction, DTDs, and variables. *Logical Methods in Computer Science*, 2(3), 2006.

[27] P. Ramanan. Efficient algorithms for minimizing tree pattern queries. In *International Conference on Management of Data (SIGMOD)*, pages 299–309, 2002.

[28] J. Robie, D. Chamberlin, M. Dyck, and J. Snelson. XML Path Language 3.0. Technical report, World Wide Web Consortium, April 2014. `www.w3.org/TR/2014/REC-xpath-30-20140408/`.

[29] J. Robie, D. Chamberlin, M. Dyck, and J. Snelson. XQuery 3.0: An XML query language. Technical report, World Wide Web Consortium, April 2014. W3C Recommendation, `www.w3.org/TR/2014/REC-xquery-30-20140408/`.

[30] I. Robinson, J. Webber, and E. Eifrem. *Graph Databases*. O'Reilly, 2 edition, 2015.

[31] M. A. Rodriguez. The Gremlin graph traversal machine and language (invited talk). In *Symposium on Database Programming Languages (DBPL)*, pages 1–10, 2015.

[32] S. Staworko and P. Wieczorek. Characterizing XML twig queries with examples. In *International Conference on Database Theory (ICDT)*, pages 144–160, 2015.

[33] SPARQL 1.1 query language. `www.w3.org/TR/sparql11-query/`. World Wide Web Consortium.

[34] Wikidata sparql query service examples. `www.wikidata.org/wiki/Wikidata:SPARQL_query_service/queries/examples`.

[35] P. T. Wood. Minimising simple XPath expressions. In *WebDB*, pages 13–18, 2001.

[36] W. Xu and Z. M. Özsoyoglu. Rewriting XPath queries using materialized views. In *International Conference on Very Large Data Bases (VLDB)*, pages 121–132, 2005.

Technical Perspective: Reflections on Extending SQL using Constraints

Surajit Chaudhuri
Microsoft Research
surajitc@microsoft.com

Relational query languages enabled the programmer to express succinctly *what* data items to retrieve using a logical model of data without any knowledge of the underlying physical structures and helped relational systems gain widespread adoption. To support applications using a relational database effectively, there has been much work subsequently along the following three dimensions:

(a) Application developers needed a programmatic way to invoke relational query functionality from within their applications. The most primitive and most prevalent form of such integration uses ODBC or JDBC APIs. While they provide connectivity to database objects, the application programmer still must manage two separate type systems and programming models. LINQ (Language Integrated Query) is an elegant example of integration where query expressions are introduced as first class citizen in the programming languages. Object-relational mapping tools allow the application programmer to continue working in their object-oriented programming paradigm even though they may be storing and retrieving relational database objects.

(b) Enriching the relational model to support extensibility so that programmers could do more within a relational database system was another key direction that has been pursued. The simplest example of such extensibility was introduction of user-defined selection and user-defined aggregates which are widely supported in relational databases. Object-relational databases extended the relational model to support complex types, inheritance and support for user-defined methods but their adoption has been relatively modest. The extensibility mechanisms in relational databases have been especially useful in adding support for richer data types such as spatial.

(c) There have been many proposals to extend core SQL by making declarative querying in relational languages to do more. The language has been extended to add recursive queries, Roll-Up, Grouping Sets, Window function and more. In addition to the extensions that have been incorporated in the SQL Standard, over time there has been a steady stream of research proposals for enriching core SQL. Adoption of any such extension requires careful consideration as they directly impact complexity of the language and the query engine.

These directions of work are largely complementary. However, there is always a healthy tension between how much should be done in applications and what is best deferred to the database server using either its extensibility mechanisms or extensions to core SQL, i.e., (b) and (c) above. Issues that influence such a debate are ease of specification and efficient execution of desired computation, data movement, and increased complexity of the database platforms.

The following paper by Brucato, Abouized, and Meliou belongs to the line of work in (c) that suggests adding more functionality to core SQL. Their work builds on past research work in the broad area of endowing the query languages with also the power of specifying constraints. The motivation for adding constraint specification to SQL comes from the desire to marry the well-established paradigms of constrained optimization e.g., Integer Linear Programming (ILP), and traditional SQL querying. Of course, selection conditions in SQL are simple row level constraints. Past work on constraint query languages proposed more generalized constraints over row values. However, the following paper (as well as a few other recent papers) focuses on *aggregate constraints* that the set of answer rows to a query must satisfy collectively, and picks the answer set based on an objective criterion (analogous to the objective function in ILP). They give a nice motivating example of formulating a meal plan for which you want to ensure that the total number of calories of the chosen meal plan is within a range and the total fat consumption is minimized. The paper lays out specific extensions to SQL needed to declaratively capture such queries and explains how such queries can be evaluated by first executing the traditional relational queries and then mapping the constraint satisfaction and objective criterion to an ILP instance which in turn can be solved using any off-the-shelf ILP solver. It is indeed advantageous to use a well-tuned off-the-shelf ILP solver in the database server using its extensibility mechanism instead of adding complexity to the core SQL engine. The rest of the paper addresses the challenges in solving large ILP problems using offline partitioning and approximation techniques to break down the global ILP instance into smaller ILP sub-problems such that the off-the-shelf ILP solvers are able to handle the scale of each sub-problem. However, proposed techniques such as offline partitioning is subject to debate as partitioning criteria for data may depend on other workload on the system such as production queries.

While the paper is unlikely to end the debate on whether or not constraints should be added to SQL (since any addition to the SQL has complex trade-offs), the paper has studied what it takes to add constraints to SQL in a relatively complete way – language extension, impact on query execution, and techniques to cope with scale. If you are interested in the topic of constraint specification and optimization over information in databases, you should definitely pay attention to this paper. It is worth a read also for any researcher who wants to consider adding extensions to core SQL to ease application tasks as it illustrates the key considerations one must address.

A Scalable Execution Engine for Package Queries

Matteo Brucato[UM] Azza Abouzied[NYU] Alexandra Meliou[UM]

[UM]College of Information and Computer Sciences
University of Massachusetts
Amherst, MA, USA
{matteo,ameli}@cs.umass.edu

[NYU]Computer Science
New York University
Abu Dhabi, UAE
azza@nyu.edu

ABSTRACT

Many modern applications and real-world problems involve the design of item collections, or *packages*: from planning your daily meals all the way to mapping the universe. Despite the pervasive need for packages, traditional data management does not offer support for their definition and computation. This is because traditional database queries follow a powerful, but very simple model: a query defines constraints that each tuple in the result must satisfy. However, a system tasked with the design of packages cannot consider items independently; rather, the system needs to determine if a set of items *collectively* satisfy given criteria.

In this paper, we present *package queries*, a new query model that extends traditional database queries to handle complex constraints and preferences over answer sets. We develop a full-fledged package query system, implemented on top of a traditional database engine. Our work makes several contributions. First, we design PaQL, a SQL-based query language that supports the declarative specification of package queries. Second, we present a fundamental strategy for evaluating package queries that combines the capabilities of databases and constraint optimization solvers. The core of our approach is a set of translation rules that transform a package query to an integer linear program. Third, we introduce an offline data partitioning strategy allowing query evaluation to scale to large data sizes. Fourth, we introduce SKETCHREFINE, an efficient and scalable algorithm for package evaluation, which offers strong approximation guarantees. Finally, we present extensive experiments over real-world data. Our results demonstrate that SKETCHREFINE is effective at deriving high-quality package results, and achieves runtime performance that is an order of magnitude faster than directly using ILP solvers over large datasets.

1. INTRODUCTION

Traditional database queries follow a simple model: they define constraints, in the form of selection predicates, that each tuple in the result must satisfy. This model is computationally efficient, as the database system can evaluate each tuple individually to determine whether it satisfies the query conditions. However, many practical, real-world problems require a collection of result tuples to satisfy constraints collectively, rather than individually.

EXAMPLE 1 (MEAL PLANNER). *A dietitian needs to design a daily meal plan for a patient. She wants a set of three gluten-free meals, between 2,000 and 2,500 calories in total, and with a low total intake of saturated fats.*

DOI: https://doi.org/10.14778/2904483.2904489

EXAMPLE 2 (NIGHT SKY). *An astrophysicist is looking for rectangular regions of the night sky that may potentially contain previously unseen quasars. Regions are explored if their overall redshift is within some specified parameters, and ranked according to their likelihood of containing a quasar [13].*

In these examples, some conditions can be verified on individual items (e.g., gluten content in a meal), while others need to be evaluated on a collection of items (e.g., total calories). Similar scenarios arise in a variety of application domains, such as investment planning, product bundles, course selection [20], team formation [2, 16], vacation and travel planning [7], and computational creativity [21]. Despite the clear application need, database systems do not currently offer support for these problems, and existing work has focused on application- and domain-specific approaches [2, 7, 16, 20, 23].

In this paper, we present a domain-independent, database-centric approach to address these challenges: We introduce a full-fledged system that supports *package queries*, a new query model that extends traditional database queries to handle complex constraints and preferences over answer sets. Package queries are defined over traditional relations, but return *packages*. A package is a collection of tuples that (a) individually satisfy *base predicates* (traditional selection predicates), and (b) collectively satisfy *global predicates* (package-specific predicates). Package queries are combinatorial in nature: the result of a package query is a (potentially infinite) set of packages, and an *objective criterion* can define a preference ranking among them.

Extending traditional database functionality to provide support for packages, rather than supporting packages at the application level, is justified by two reasons: First, the features of packages and the algorithms for constructing them are not unique to each application; therefore, the burden of package support should be lifted off application developers, and database systems should support package queries like traditional queries. Second, the data used to construct packages typically reside in a database system, and packages themselves are structured data objects that should naturally be stored in and manipulated by a database system.

Our work addresses *three important challenges*. The first challenge is to support *declarative* specification of packages. SQL enables the declarative specification of properties that result tuples should satisfy. In Example 1, it is easy to specify the exclusion of meals with gluten using a regular selection predicate in SQL. However, it is difficult to specify global constraints (e.g., total calories of a set of meals should be between 2,000 and 2,500 calories). Expressing such a query in SQL requires either complex self-joins that explode the size of the query, or recursion, which results in extremely complex queries that are hard to specify and optimize. Our goal is to maintain the declarative power of SQL, while extending its expressiveness to allow for the easy specification of packages.

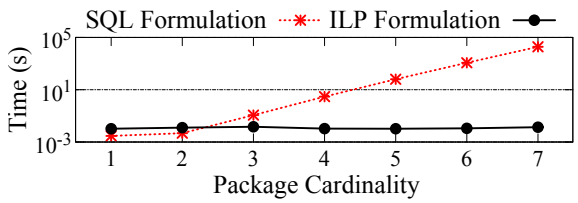


Figure 1: Traditional database technology is ineffective at package evaluation, and the runtime of a SQL formulation of a package query grows exponentially. In contrast, tools such as ILP solvers are more effective.

The second challenge relates to the *evaluation* of package queries. Due to their combinatorial complexity, package queries are harder to evaluate than traditional database queries [8]. Package queries are in fact as hard as integer linear programs [4]. Existing database technology is ineffective at evaluating package queries, even if one were to express them in SQL. Figure 1 shows the performance of evaluating a package query expressed as a multi-way self-join query in traditional SQL. As the cardinality of the package increases, so does the number of joins, and the runtime quickly becomes prohibitive: In a small set of 100 tuples from the Sloan Digital Sky Survey dataset [22], SQL evaluation takes almost 24 hours to construct a package of 7 tuples. Our goal is to extend the database evaluation engine to take advantage of external tools, such as ILP solvers, which are more effective for combinatorial problems.

The third challenge pertains to query evaluation *performance* and *scaling* to large datasets. Integer programming solvers have two major limitations: they require the entire problem to fit in main memory, and they fail when the problem is too complex (e.g., too many variables and/or too many constraints). Our goal is to overcome these limitations through sophisticated evaluation methods that allow solvers to scale to large data sizes.

Our work addresses these challenges through the design of language and algorithmic support for the specification and evaluation of package queries. We present PaQL (Package Query Language), a declarative language that provides simple extensions to standard SQL to support constraints at the package level. PaQL is at least as expressive as integer linear programming, which implies that evaluation of package queries is NP-hard [4]. We present a fundamental evaluation strategy, DIRECT, that combines the capabilities of databases and constraint optimization solvers to derive solutions to package queries. The core of our approach is a set of translation rules that transform a package query to an integer linear program. This translation allows for the use of highly-optimized external solvers for the evaluation of package queries. We introduce an offline data partitioning strategy that allows package query evaluation to scale to large data sizes. The core of our evaluation strategy, SKETCHREFINE, lies on separating the package computation into multiple stages, each with small subproblems, which the solver can evaluate efficiently. In the first stage, the algorithm "sketches" an initial sample package from a set of representative tuples, while the subsequent stages "refine" the current package by solving an ILP within each partition. SKETCHREFINE offers strong approximation guarantees for the package results compared to DIRECT. We present an extensive experimental evaluation on real-world data that shows that our query evaluation method SKETCHREFINE: (1) is able to produce packages an order of magnitude faster than the ILP solver used directly on the entire problem; (2) scales up to sizes that the solver cannot manage directly; (3) produces packages of very good quality in terms of objective value.

2. LANGUAGE SUPPORT FOR PACKAGES

Data management systems do not natively support package queries. While there are ways to express package queries in SQL, these are cumbersome and inefficient.

Specifying packages with self-joins. When packages have strict cardinality (number of tuples), and only in this case, it is possible to express package queries using traditional self-joins. For instance, self-joins can express the query of Example 1 as follows:

```
SELECT *   FROM Recipes R1, Recipes R2, Recipes R3
WHERE      R1.pk < R2.pk AND R2.pk < R3.pk AND
  R1.gluten = 'free' AND R2.gluten = 'free' AND R3.gluten = 'free'
  AND R1.kcal + R2.kcal + R3.kcal BETWEEN 2.0 AND 2.5
ORDER BY  R1.saturated_fat + R2.saturated_fat + R3.saturated_fat
```

This query is efficient only for constructing packages with very small cardinality: larger cardinality requires a larger number of self-joins, quickly rendering evaluation time prohibitive (Figure 1). The benefit of this specification is that the optimizer can use the traditional relational algebra operators, and augment its decisions with package-specific strategies. However, this method does not apply for packages of unbounded cardinality.

Using recursion in SQL. More generally, SQL can express package queries by generating and testing each possible subset of the input relation. This requires recursion to build a *powerset table*; checking each set in the powerset table for the query conditions will yield the result packages. This approach has three major drawbacks. First, it is not declarative, and the specification is tedious and complex. Second, it is not amenable to optimization in existing systems. Third, it is extremely inefficient to evaluate, because the powerset table generates an exponential number of candidates.

2.1 *PaQL*: The Package Query Language

Our goal is to support package specification in a declarative and intuitive way. In this section, we describe PaQL, a declarative query language that introduces simple extensions to SQL to define package semantics and package-level constraints. We first show how PaQL can express the query of Example 1, as our running example, to demonstrate the new language features:

```
Q: SELECT       PACKAGE(R) AS P
   FROM         Recipes R REPEAT 0
   WHERE        R.gluten = 'free'
   SUCH THAT    COUNT(P.*) = 3 AND
                SUM(P.kcal) BETWEEN 2.0 AND 2.5
   MINIMIZE     SUM(P.saturated_fat)
```

Basic semantics. The new keyword PACKAGE differentiates PaQL from traditional SQL queries.

```
Q1: SELECT  *              Q2: SELECT  PACKAGE(R) AS P
    FROM    Recipes R          FROM    Recipes R
```

The semantics of $\mathcal{Q}_1$ and $\mathcal{Q}_2$ are fundamentally different: $\mathcal{Q}_1$ is a traditional SQL query, with a unique, finite result set (the entire Recipes table), whereas there are infinitely many packages that satisfy the package query $\mathcal{Q}_2$: all possible *multisets* of tuples from the input relation. The result of a package query like $\mathcal{Q}_2$ is a set of packages. Each package resembles a relational table containing a collection of tuples (with possible repetitions) from relation Recipes, and therefore a package result of $\mathcal{Q}_2$ follows the schema of Recipes.

The specification of $\mathcal{Q}_2$ allows for arbitrary repetitions of tuples, thus, there are infinitely many packages that satisfy the query. Although semantically valid, a query like $\mathcal{Q}_2$ would not occur in practice, as most application scenarios expect few, or even exactly

one result. We proceed to describe the additional constraints in the example query $\mathcal{Q}$ that restrict the number of package results.

Repetition constraint. The REPEAT 0 statement in query $\mathcal{Q}$ specifies that no tuple from the input relation can appear multiple times in a package result. If this restriction is absent (as in query $\mathcal{Q}_2$), tuples can be repeated an unlimited number of times. By allowing no repetitions, $\mathcal{Q}$ restricts the package space from infinite to 2^n, where n is the size of the input relation. Generalizing, the specification REPEAT $\mathcal{K}$ allows a package to repeat tuples up to $\mathcal{K}$ times, resulting in $(2+\mathcal{K})^n$ candidate packages.

Base and global predicates. A package query defines two types of predicates. A *base predicate*, defined in the WHERE clause, is equivalent to a selection predicate and can be evaluated with standard SQL: any tuple in the package needs to *individually* satisfy the base predicate. For example, query $\mathcal{Q}$ specifies the base predicate: R.gluten = 'free'. Since base predicates directly filter input tuples, they are specified over the input relation R. *Global predicates* are the core of package queries, and they appear in the new SUCH THAT clause. Global predicates are higher-order than base predicates: they cannot be evaluated on individual tuples, but on tuple collections. Since they describe package-level constraints, they are specified over the package result P, e.g., COUNT(P.$*$) = 3, which limits the query results to packages of exactly 3 tuples.

The global predicates shown in query $\mathcal{Q}$ abbreviate aggregates that are in reality subqueries. For example, COUNT(P.$*$) = 3, is an abbreviation for (SELECT COUNT($*$) FROM P) = 3. Using subqueries, PaQL can express arbitrarily complex global constraints among aggregates over a package.

Objective clause. The objective clause specifies a ranking among candidate package results, and appears with either the MINIMIZE or MAXIMIZE keyword. It is a condition on the package-level, and hence it is specified over the package result P, e.g., MINIMIZE SUM(P.saturated_fat). Similarly to global predicates, this form is a shorthand for MINIMIZE (SELECT SUM(saturated_fat) FROM P). A PaQL query with an objective clause returns a single result: the package that optimizes the value of the objective. The evaluation methods that we present in this work focus on such queries. In prior work [5], we described preliminary techniques for returning multiple packages in the absence of optimization objectives, but a thorough study of such methods is left to future work.

Expressiveness and complexity. PaQL can express general integer linear programs, which means that evaluation of package queries is NP-complete [4]. As a first step in package evaluation, we proceed to show how a PaQL query can be transformed into a linear program and solved using general ILP solvers.

3. ILP FORMULATION

In this section, we present an ILP formulation for package queries, which is at the core of our evaluation methods DIRECT and SKETCHREFINE. The results in this section are inspired by the translation rules employed by Tiresias [17] to answer *how-to queries*.

3.1 PaQL to ILP Translation

Let R indicate the input relation, $n = |\mathsf{R}|$ the number of tuples in R, R.attr an attribute of R, P a package, f a linear aggregate function (such as COUNT and SUM), $\odot \in \{\leq, \geq\}$ a constraint inequality, and $v \in \mathbb{R}$ a constant. For each tuple t_i from R, $1 \leq i \leq n$, the ILP problem includes a nonnegative integer variable x_i ($x_i \geq 0$), indicating the number of times t_i is included in an answer package. We also use $\bar{x} = \langle x_1, x_2, \ldots, x_n \rangle$ to denote the vector of all integer variables. A PaQL query is formulated as an ILP problem using the following translation rules:

Repetition constraint. The REPEAT keyword, expressible in the FROM clause, restricts the domain that the variables can take on. Specifically, REPEAT $\mathcal{K}$ implies $0 \leq x_i \leq \mathcal{K}+1$.

Base predicate. Let β be a base predicate, e.g., R.gluten = 'free', and R_β the *base relation* containing tuples from R satisfying β. We encode β by setting $x_i = 0$ for every tuple $t_i \notin \mathsf{R}_\beta$.

Global predicate. Each global predicate in the SUCH THAT clause takes the form $f(\mathsf{P}) \odot v$. For each such predicate, we derive a linear function $f'(\bar{x})$ over the integer variables. A cardinality constraint $f(\mathsf{P}) = \mathsf{COUNT}(\mathsf{P}.*)$ is linearly translated into $f'(\bar{x}) = \sum_i x_i$. A summation constraint $f(\mathsf{P}) = \mathsf{SUM}(\mathsf{P}.\mathsf{attr})$ is linearly translated into $f'(\bar{x}) = \sum_i (t_i.\mathsf{attr}) x_i$. Other non-trivial constraints and general Boolean expressions over the global predicates can be encoded into a linear program with the help of Boolean variables and linear transformation tricks found in the literature [3]. We refer to the original version of this paper for further details [4].

Objective clause. We encode MAXIMIZE $f(\mathsf{P})$ as $\max f'(\bar{x})$, where $f'(\bar{x})$ is the encoding of $f(\mathsf{P})$. Similarly MINIMIZE $f(\mathsf{P})$ is encoded as $\min f'(\bar{x})$. If the query does not include an objective clause, we add the *vacuous* objective $\max \sum_i 0 \cdot x_i$.

3.2 Query Evaluation with DIRECT

Using the ILP formulation, we develop our basic evaluation method for package queries, called DIRECT. We later extend this technique to our main algorithm, SKETCHREFINE, which supports efficient package evaluation in large data sets.

Package evaluation with DIRECT employs three simple steps:

1. **ILP formulation.** We transforms a PaQL query to an ILP problem using the rules described in Section 3.1.
2. **Base relation.** We compute the base relation R_β with a traditional SQL query that selects tuples from R that satisfy the base predicate. After this phase, all variables x_i such that $x_i = 0$ can be eliminated from the ILP problem.
3. **ILP execution.** We employ an off-the-shelf ILP solver, as a black box, to get a solution x_i^* for all the integer variables x_i of the problem. Each x_i^* informs the number of times tuple t_i should be included in the answer package.

The DIRECT algorithm has two crucial *drawbacks*. First, it is only applicable if the input relation is small enough to fit entirely in main memory: ILP solvers, such as IBM's CPLEX, require the entire problem to be loaded in memory before execution. Second, even for problems that fit in main memory, this approach may fail due to the complexity of the integer problem. In fact, integer linear programming is a notoriously hard problem, and modern ILP solvers use algorithms, such as *branch-and-cut* [19], that often perform well in practice, but can "choke" even on small problem sizes due to their exponential worst-case complexity [6]. This may result in unreasonable performance due to solvers using too many resources (main memory, virtual memory, CPU time), eventually thrashing the entire system.

4. SCALABLE PACKAGE EVALUATION

In this section, we present SKETCHREFINE, an approximate divide-and-conquer technique for efficiently answering package queries on large datasets. SKETCHREFINE smartly decomposes a query into smaller queries, formulates them as ILP problems, and employs an ILP solver as a black-box component to answer each individual query. By breaking down the problem into smaller subproblems, the algorithm avoids the drawbacks of the DIRECT approach.

The algorithm is based on an important observation: *similar tuples are likely to be interchangeable within packages*. A group of

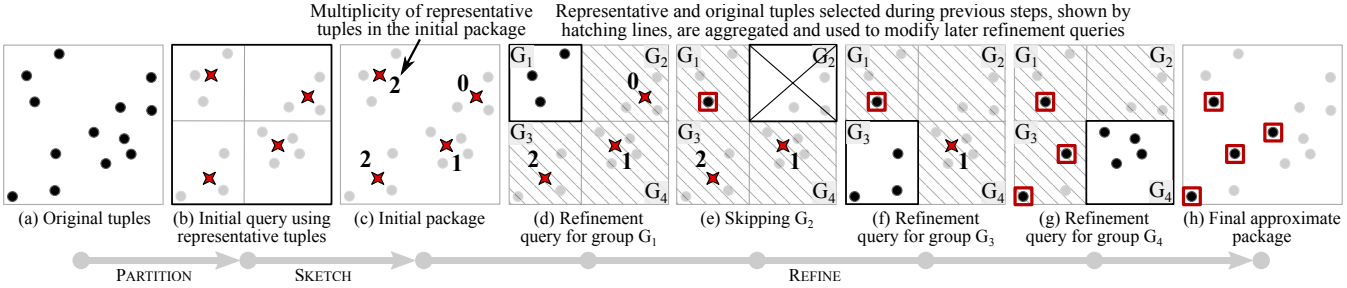


Figure 2: The original tuples (a) are partitioned into four groups and a representative is constructed for each group (b). The initial sketch package (c) contains only representative tuples, with possible repetitions up the size of each group. The refine query for group G_1 (d) involves the original tuples from G_1 and the aggregated solutions to all other groups (G_2, G_3, and G_4). Group G_2 can be skipped (e) because no representatives could be picked from it. Any solution to previously refined groups are used while refining the solution for the remaining groups (f and g). The final approximate package (h) contains only original tuples.

similar tuples can therefore be "compressed" to a single *representative tuple* for the entire group. SKETCHREFINE *sketches* an initial answer package using only the set of representative tuples, which is substantially smaller than the original dataset. This initial solution is then *refined* by evaluating a subproblem for each group, iteratively replacing the representative tuples in the current package solution with original tuples from the dataset. Figure 2 provides a high-level illustration of the three main steps of SKETCHREFINE:

1. **Offline partitioning (Section 4.1).** The algorithm assumes a partitioning of the data into groups of similar tuples. This partitioning is performed offline (not at query time). In our implementation, we partition data using k-dimensional quad trees [9], but other partitioning schemes are possible.
2. **Sketch (Section 4.2.1).** SKETCHREFINE sketches an initial package by evaluating the package query only over the set of representative tuples.
3. **Refine (Section 4.2.2).** Finally, SKETCHREFINE transforms the initial package into a complete package by replacing each representative tuple with some of the original tuples from the same group, one group at a time.

SKETCHREFINE always constructs *feasible* packages, i.e., packages that satisfy all the query constraints, but with a possibly suboptimal objective value. However, SKETCHREFINE offers strong approximation guarantees compared to the solution generated by DIRECT for the same query. SKETCHREFINE may suffer from *false infeasibility*, which happens when the algorithm reports a feasible query to be infeasible. The probability of false infeasibility is, however, low and bounded. We formalize these properties in Section 4.3.

In the subsequent discussion, we use R to denote the input relation of n tuples, $t_i \in \mathsf{R}$, $1 \leq i \leq n$. R is partitioned into m groups $G_1, \ldots, G_m$. Each group G_j, $1 \leq j \leq m$, has a representative tuple $\tilde{t}_j$, which may not always appear in R. We denote the partitioned space with $\mathcal{P} = \{(G_j, \tilde{t}_j) \mid 1 \leq j \leq m\}$. We refer to packages that contain some representative tuples as *sketch packages* and packages with only original tuples as *complete packages* (or simply *packages*). We denote a complete package with p and a sketch package with $p_{\mathcal{S}}$, where $\mathcal{S} \subseteq \mathcal{P}$ is the set of groups that are yet to be refined to transform $p_{\mathcal{S}}$ into a complete answer package p.

4.1 Offline Partitioning

SKETCHREFINE relies on an offline partitioning of the input relation R into groups of similar tuples. Partitioning is based on a set of k numerical *partitioning attributes*, $\mathcal{A}$, from the input relation R, and uses two parameters: a *size threshold* and (optionally) a *radius limit*. The size threshold τ, $1 \leq \tau \leq n$, restricts the size of each partitioning group G_j, $1 \leq j \leq m$, to a maximum of τ original tuples, i.e., $|G_j| \leq \tau$. The *radius* $r_j \geq 0$ of a group G_j is the greatest absolute distance between the representative tuple of G_j, $\tilde{t}_j$, and every original tuple of the group, across all partitioning attributes. The radius limit ω, $\omega \geq 0$, requires that for every partitioning group G_j, $1 \leq j \leq m$, $r_j \leq \omega$.

Setting the partitioning parameters. The size threshold, τ, affects the number of clusters, m, as smaller clusters (lower τ) imply more of them (larger m), especially on skewed datasets. For best response time of SKETCHREFINE, τ should be set so that both m and τ are small. Our experiments show that a proper setting can yield an order of magnitude improvement in query response time.

The *optional* radius limit, ω, helps ensure that a result produced by SKETCHREFINE is within a guaranteed approximation bound from the package that DIRECT would generate. Enforcing a radius limit requires more partitioning iterations, which increases the cost of offline partitioning. However, our experiments show that even without enforcing an approximation guarantee, SKETCHREFINE produces satisfactory answers.

Partitioning method. Our partitioning procedure is based on k-dimensional *quad-tree indexing* [9]. The method recursively partitions a relation into groups until all the groups satisfy the size threshold and meet the radius limit. The procedure initially creates a single group G_1 that includes all the original tuples from relation R. Our method recursively computes the sizes and radii of the current groups, as well as the *centroid* of each group. It then partitions the groups that violate either the size or the radius limits, using the centroids as partitioning boundaries. In the last iteration, the centroids for each group become the representative tuples, $\tilde{t}_j$, $1 \leq j \leq m$, and get stored in a new *representative relation* $\tilde{\mathsf{R}}(\mathsf{gid}, \mathsf{attr}_1, \ldots, \mathsf{attr}_k)$.

One-time cost. Partitioning is an expensive procedure. To avoid paying its cost at query time, the dataset is partitioned in advance and used to answer a workload of package queries. In order to ensure the approximation guarantees, the partitioning attributes, $\mathcal{A}$, must be a superset of the query attributes. For a known workload, our experiments show that partitioning the dataset on the union of all query attributes provides the best performance in terms of query evaluation time and approximation error for the computed answer package. We also demonstrate that our query evaluation approach is robust to a wide range of partition sizes, and to imperfect partitions that cover more or fewer attributes than those used in a particular query [4]. This means that, even without a known workload, a partitioning performed on all of the data attributes still provides good performance. Note that the same partitioning can be used to support a multitude of queries over the same dataset. In our experiments, we show that a single partitioning performs consistently well across different queries.

4.2 Query Evaluation with SketchRefine

During query evaluation, SketchRefine first *sketches* a package solution using the representative tuples (Sketch), and then it *refines* it by replacing representative tuples with original tuples (Refine). We describe these steps using the example query $\mathcal{Q}$ from Section 2.1.

4.2.1 Sketch

Using the representative relation $\tilde{\mathsf{R}}$ produced by the partitioning, the Sketch procedure constructs and evaluates a *sketch query*, $\mathcal{Q}[\tilde{\mathsf{R}}]$. The result is an initial sketch package, $p_{\mathcal{S}}$, containing representative tuples that satisfy the same constraints as the original query $\mathcal{Q}$:

```
Q[R̃]: SELECT      PACKAGE(R̃) AS p_S
      FROM        R̃
      WHERE       R̃.gluten = 'free'
      SUCH THAT
        COUNT(p_S.*) = 3 AND
        SUM(p_S.kcal) BETWEEN 2.0 AND 2.5 AND
        (SELECT COUNT(*) FROM p_S WHERE gid = 1) ≤ |G_1|
        AND ...
        (SELECT COUNT(*) FROM p_S WHERE gid = m) ≤ |G_m|
      MINIMIZE    SUM(p_S.saturated_fat)
```

The new global constraints (in bold) ensure that every representative tuple does not appear in $p_{\mathcal{S}}$ more times than the size of its group, G_j. This accounts for the repetition constraint REPEAT 0 in the original query. Generalizing, with REPEAT $\mathcal{K}$, each $\tilde{t}_j$ can be repeated up to $|G_j|(1+\mathcal{K})$ times. These constraints are omitted from $\mathcal{Q}[\tilde{\mathsf{R}}]$ if the original query does not contain a repetition constraint.

Since the representative relation $\tilde{\mathsf{R}}$ contains exactly m representative tuples, the ILP problem corresponding to this query has only m variables. This is typically small enough for the black box ILP solver to manage directly, and thus we can solve this package query using the Direct method. If m is too large, we can solve this query *recursively* with SketchRefine: the set of m representatives is further partitioned into smaller groups until the subproblems reach a size that can be efficiently solved directly.

The Sketch procedure *fails* if the sketch query $\mathcal{Q}[\tilde{\mathsf{R}}]$ is infeasible, in which case SketchRefine reports the original query $\mathcal{Q}$ as infeasible. This may constitute *false infeasibility*, if $\mathcal{Q}$ is actually feasible. However, we show that the probability of false infeasibility is low and bounded (Section 4.3).

4.2.2 Refine

Using the sketched solution over the representative tuples, the Refine procedure iteratively replaces the representative tuples with tuples from the original relation R, until no more representatives are present in the package. The algorithm *refines* the sketch package $p_{\mathcal{S}}$, one group at a time: For a group G_j with representative $\tilde{t}_j \in p_{\mathcal{S}}$, the algorithm derives package $\bar{p}_j$ from $p_{\mathcal{S}}$ by eliminating all instances of $\tilde{t}_j$; it then seeks to replace the eliminated representatives with actual tuples, by issuing a *refine query*, $\mathcal{Q}[G_j]$, on group G_j:

```
Q[G_j]: SELECT      PACKAGE(G_j) AS p_j
        FROM        G_j REPEAT 0
        WHERE       G_j.gluten = 'free'
        SUCH THAT
          COUNT(p_j.*) + COUNT(p̄_j.*) = 3 AND
          SUM(p_j.kcal) + SUM(p̄_j.kcal) BETWEEN 2.0 AND 2.5
        MINIMIZE    SUM(p_j.saturated_fat)
```

The query derives a set of tuples p_j, as a replacement for the occurrences of the representatives of G_j in $p_{\mathcal{S}}$. The global constraints in $\mathcal{Q}[G_j]$ ensure that the combination of tuples in p_j and $\bar{p}_j$ satisfy the original query $\mathcal{Q}$. Thus, this step produces the new *refined sketch package* $p'_{\mathcal{S}'} = \bar{p}_j \cup p_j$, where $\mathcal{S}' = \mathcal{S} \setminus \{(G_j, \tilde{t}_j)\}$.

Since G_j has at most τ tuples, the ILP problem corresponding to $\mathcal{Q}[G_j]$ has at most τ variables. This is typically small enough for the black box ILP solver to solve directly, and thus we can solve this package query using the Direct method. Similarly to the sketch query, if τ is too large, we can solve this query recursively with SketchRefine: the tuples in group G_j are further partitioned into smaller groups until the subproblems reach a size that can be efficiently solved directly.

Ideally, the Refine step will only process each group with representatives in the initial sketch package once. However, the order of refinement matters, as each refinement step is greedy: it selects tuples to replace the representatives of a single group, without considering the effects of this choice on other groups. As a result, a particular refinement step may render the query infeasible (no tuples from the remaining groups can satisfy the constraints). When this occurs, Refine employs a *greedy backtracking* strategy that reconsiders groups in a different order.

Greedy backtracking. Refine activates backtracking when it encounters an infeasible *refine query*, $\mathcal{Q}[G_j]$. Backtracking *greedily prioritizes* the infeasible groups. This choice is motivated by a simple heuristic: if the refinement on G_j fails, it is likely due to choices made by previous refinements; therefore, by prioritizing G_j, we reduce the impact of other groups on the feasibility of $\mathcal{Q}[G_j]$. This heuristic does not affect the approximation guarantees.

The algorithm logically traverses a *search tree* (which is only constructed as new branches are created and new nodes visited), where each node corresponds to a unique sketch package $p_{\mathcal{S}}$. The traversal starts from the *root*, corresponding to the initial sketch package, where no groups have been refined ($\mathcal{S} = \mathcal{P}$), and finishes at the first encountered *leaf*, corresponding to a complete package ($\mathcal{S} = \emptyset$). The algorithm terminates as soon as it encounters a complete package, which it returns. The algorithm assumes a (initially random) refinement order for all groups in $\mathcal{S}$, and places them in a priority queue. During refinement, this group order can change by prioritizing groups with infeasible refinements.

Run time complexity. In the best case, all refine queries are feasible and the algorithm never backtracks. In this case, the algorithm makes up to m calls to the ILP solver to solve problems of size up to τ, one for each refining group. In the worst case, SketchRefine tries every group ordering leading to an exponential number of calls to the ILP solver. Our experiments show that the best case is the most common and backtracking occurs infrequently.

4.3 Theoretical Guarantees

We present two important results on the theoretical guarantees of SketchRefine: (1) it produces packages that closely approximate the objective value of the packages produced by Direct, and (2) the probability of false negatives (i.e., queries incorrectly deemed infeasible) is low and bounded.

We prove that for a desired approximation parameter ε, we can derive a radius limit ω for the offline partitioning that guarantees that SketchRefine will produce a package with objective value $(1 \pm \varepsilon)^6$-factor close to the objective value of the solution generated by Direct for the same query.

Theorem 1 (Approximation Bounds). *For any feasible package query with a maximization (minimization, resp.) objective and approximation parameter ε, $0 \le \varepsilon < 1$ ($\varepsilon \ge 0$, resp.), any database instance, any set of partitioning attributes $\mathcal{A}$, superset of the numerical query attributes, any size threshold τ, and radius limit:*

$$\omega = \min_{\substack{1 \le j \le m \\ \mathsf{attr} \in \mathcal{A}}} \gamma\, |\tilde{t}_j.\mathsf{attr}|, \quad \textit{where } \gamma = \varepsilon \quad (\gamma = \tfrac{\varepsilon}{1+\varepsilon},\ \textit{resp.}) \qquad (1)$$

The package produced by SKETCHREFINE *(if any) is guaranteed to have objective value* $\geq (1-\varepsilon)^6 OPT$ *(*$\leq (1+\varepsilon)^6 OPT$*, resp.), where OPT is the objective value of the* DIRECT *solution.*

For a feasible query $\mathcal{Q}$, *false infeasibility* may happen in two cases: (1) when the sketch query $\mathcal{Q}[\tilde{\mathsf{R}}]$ is infeasible; (2) when greedy backtracking fails (possibly due to suboptimal partitioning). In both cases, SKETCHREFINE would (incorrectly) report a feasible package query as infeasible. False negatives are, however, extremely rare, as the following theorem establishes.

THEOREM 2 (FALSE INFEASIBILITY). *For any feasible package query, any database instance, any set of partitioning attributes* $\mathcal{A}$ *that is a superset of the query attributes, any size threshold* τ*, and any radius limit* ω*,* SKETCHREFINE *finds a feasible package with high probability that inversely depends on query selectivity.*

5. EXPERIMENTAL EVALUATION

We present an extensive experimental evaluation of our techniques for package queries on real-world data. Our results show the following properties of our methods: (1) SKETCHREFINE evaluates package queries an order of magnitude faster than DIRECT; (2) SKETCHREFINE scales up to sizes that DIRECT cannot handle directly; (3) SKETCHREFINE produces packages of high quality (similar objective value as the packages returned by DIRECT). We have also performed extensive experiments on benchmark data and have investigated the effects of imperfect partitioning over different sets of attributes, demonstrating the robustness of SKETCHREFINE under these variations [4].

5.1 Experimental Setup

We implemented our package evaluation system as a layer on top of PostgreSQL. The system interacts with the DBMS via SQL. A package is materialized into the DBMS, as a relation, only when necessary (for example, to compute its objective value). We employ IBM's CPLEX [12] as our black-box ILP solver. We compare DIRECT with SKETCHREFINE. Both methods use the PaQL to ILP translation presented in Section 3.1: DIRECT translates and solves the original query; SKETCHREFINE translates and solves the subqueries. We demonstrate the performance of our query evaluation methods using a real-world dataset consisting of approximately 5.5 million tuples extracted from the Galaxy view of the Sloan Digital Sky Survey (SDSS) [22]. We constructed a set of seven package queries, by adapting some of the real-world sample SQL queries available directly from the SDSS website.

We evaluate methods on their *efficiency* (response time) and *effectiveness* (approximation ratio):

Response time: The wall-clock time to generate an answer package. This only includes the time to translate the PaQL query into one or several ILP problems, the time to load the problems into the solver, and the time taken by the solver to produce a solution.

Approximation ratio: We compare the objective value of a package returned by SKETCHREFINE with the objective value of the package returned by DIRECT on the same query. Using Obj_S and Obj_D to denote the objective values of SKETCHREFINE and DIRECT, respectively, we compute the empirical *approximation ratio* $\frac{Obj_D}{Obj_S}$ for maximization queries, and $\frac{Obj_S}{Obj_D}$ for minimization queries. An approximation ratio of one indicates that SKETCHREFINE produces a solution with same objective value as the solution produced by the solver on the entire problem. The higher the approximation ratio, the lower the quality of the result package.

5.2 Results and Discussion

We evaluate two fundamental aspects of our algorithms: (1) their query response time and approximation ratio with increasing dataset sizes; (2) the impact of varying partitioning size thresholds (τ) on SKETCHREFINE's performance. Further, our analysis has shown that SKETCHREFINE is robust to imperfect partitioning [4].

5.2.1 Query performance as data set size increases

In our first set of experiments, we evaluate the scalability of our methods on input relations of increasing size. First, we partition each dataset using the union of all package query attributes in the workload: we refer to these partitioning attributes as the *workload attributes*. We do not enforce a radius condition (ω) during partitioning for two reasons: (1) to show that an offline partitioning can be used to answer efficiently and effectively both maximization and minimization queries, even though they would normally require different radii; (2) to demonstrate the effectiveness of SKETCHREFINE in practice, even without having theoretical guarantees in place.

We perform offline partitioning with partition size threshold τ set to 10% of the dataset size and without a radius limit. We derive the partitionings for the smaller data sizes (less than 100% of the dataset), by randomly removing tuples from the original partitions. This operation is guaranteed to maintain the size condition.

Figure 3 reports our scalability results on the Galaxy workload. The figure displays the query runtimes in seconds on a logarithmic scale, averaged across 10 runs for each datapoint. At the bottom of each figure, we also report the mean and median approximation ratios across all dataset sizes. The graph for Q2 does not report approximation ratios because DIRECT evaluation fails to produce a solution for this query across all data sizes. We observe that DIRECT can scale up to millions of tuples in three of the seven queries. Its run-time performance degrades, as expected, when data size increases, but even for very large datasets DIRECT is usually able to answer the package queries in less than a few minutes. However, DIRECT has high failure rate for some of the queries, indicated by the missing data points in some graphs (queries Q2, Q3, Q6 and Q7). This happens when CPLEX uses the entire available main memory while solving the corresponding ILP problems. For some queries, such as Q3 and Q7, this occurs with bigger dataset sizes. However, for queries Q2 and Q6, DIRECT even fails on small data. This is a clear demonstration of one of the major limitations of ILP solvers: they can fail even when the dataset can fit in main memory, due to the complexity of the integer problem. In contrast, our scalable SKETCHREFINE algorithm is able to perform well on all dataset sizes and across all queries. SKETCHREFINE consistently performs about an order of magnitude faster than DIRECT across all queries. Its running time is consistently below one or two minutes, even when constructing packages from millions of tuples.

Both the mean and median approximation ratios are very low, usually all close to one or two. This shows that the substantial gain in running time of SKETCHREFINE over DIRECT does not compromise the quality of the resulting packages. Our results indicate that the overhead of partitioning with a radius condition is often unnecessary in practice. Since the approximation ratio is not enforced, SKETCHREFINE can potentially produce bad solutions, but this happens rarely.

5.2.2 Effect of varying partition size threshold

In our second set of experiments, we vary τ, which is used during partitioning to limit the size of each partition, to study its effects on the query response time and the approximation ratio of SKETCHREFINE. In all cases, along the lines of the previous experiments, we do not enforce a radius condition. Figure 4 show the results obtained

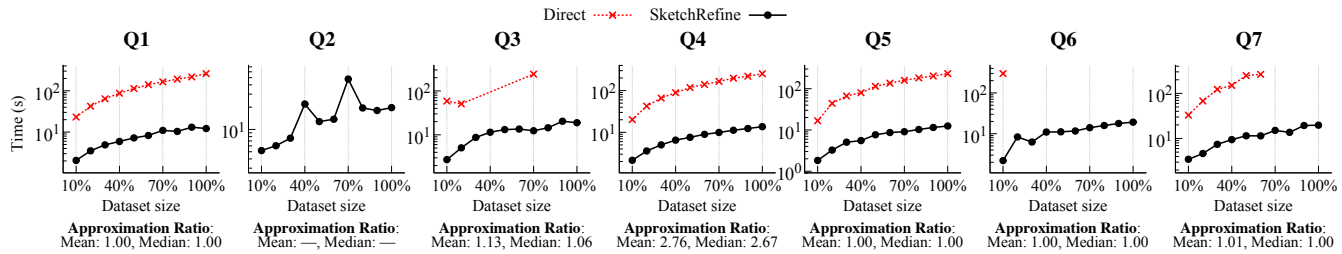


Figure 3: Scalability on the Galaxy workload. SKETCHREFINE uses an offline partitioning computed on the full dataset, using the workload attributes, $\tau = 10\%$ of the dataset size, and no radius condition. DIRECT scales up to millions of tuples in about half of the queries, but it fails on the other half. SKETCHREFINE scales up nicely in all cases, and runs about an order of magnitude faster than DIRECT. Its approximation ratio is always low, even though the partitioning is constructed without radius condition.

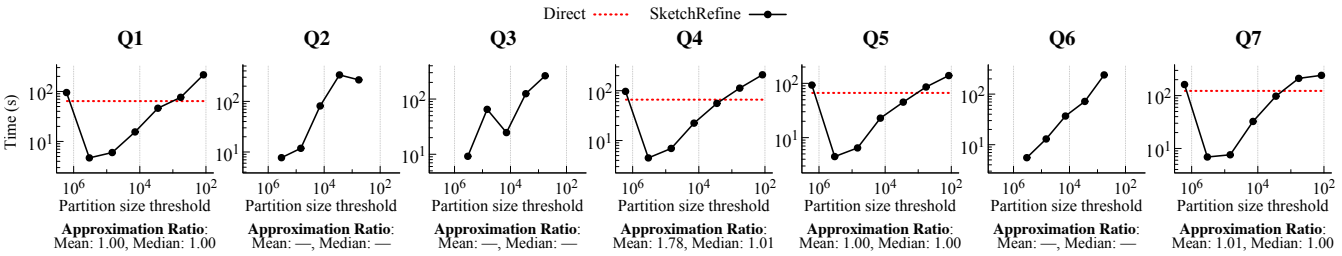


Figure 4: Impact of partition size threshold τ on the Galaxy workload, using 30% of the original dataset. Partitioning is performed at each value of τ using all the workload attributes, and with no radius condition. The baseline DIRECT and the approximation ratios are only shown when DIRECT is successful. The results show that τ has a major impact on the running time of SKETCHREFINE, but almost no impact on the approximation ratio. DIRECT can be an order of magnitude faster than DIRECT with proper tuning of τ.

on the Galaxy workload, using 30% of the original data. We vary τ from higher values corresponding to fewer but larger partitions, on the left-hand size of the *x*-axis, to lower values, corresponding to more but smaller partitions. When DIRECT is able to produce a solution, we also report its running time (horizontal line) as a baseline for comparison.

Our results show that the partition size threshold has a major impact on the execution time of SKETCHREFINE, with extreme values of τ (either too low or too high) often resulting in slower running times than DIRECT. With bigger partitions, on the left-hand side of the *x*-axis, SKETCHREFINE takes about the same time as DIRECT because both algorithms solve problems of comparable size. When the size of each partition starts to decrease, moving from left to right on the *x*-axis, the response time of SKETCHREFINE decreases rapidly, reaching about an order of magnitude improvement with respect to DIRECT. Most of the queries show that there is a "sweet spot" at which the response time is the lowest: when all partitions are small, and there are not too many of them. The point is consistent across different queries, showing that it only depends on the input data size. After that point, although the partitions become smaller, the number of partitions starts to increase significantly. This increase has two negative effects: it increases the number of representative tuples, and thus the size and complexity of the initial sketch query, and it increases the number of groups that REFINE may need to refine to construct the final package. This causes the running time of SKETCHREFINE, on the right-hand side of the *x*-axis, to increase again and reach or surpass the running time of DIRECT. The mean and median approximation ratios are in all cases very close to one, indicating that SKETCHREFINE retains very good quality regardless of the partition size threshold.

6. RELATED WORK

We discuss related work from the following areas: package recommendation systems, semantic window queries, how-to queries, constraint query languages, and approximation techniques for ILP formulations and subclasses of package queries.

Package or set-based *recommendation systems* are closely related to package queries. A package recommendation system presents users with interesting sets of items that satisfy some global conditions. Specific application scenarios usually drive these systems. For instance, in the CourseRank [20] system, the items to be recommended are university *courses*, and the types of constraints are course-specific (e.g., prerequisites, incompatibilities, etc.). *Satellite packages* [1] are sets of items, such as smartphone accessories, that are compatible with a "central" item, such as a smartphone. Other related problems in the area of package recommendations are *team formation* [16, 2], and recommendation of *vacation* and *travel packages* [7]. Queries expressible in these frameworks are also expressible in PaQL, but the opposite does not hold. The complexity of set-based package recommendation problems is studied in [8], where the authors show that computing top-*k* packages with a conjunctive query language is harder than NP-complete.

Packages are also related to the *semantic windows* [13] expressible in Searchlight [14]. A semantic window defines a contiguous subset of a grid-partitioned space with certain global properties. These queries can be expressed in PaQL by adding global constraints that ensure contiguity in the grid. Packages, however, are more general than semantic windows because they allow regions to be non-contiguous or contain gaps. Searchlight has several other major differences with our work: (1) it computes optimal solutions by enumerating the feasible ones and retaining the optimal, whereas our methods do not require enumeration; (2) it assumes that the solver

implements redundant and arbitrary data access paths while solving the problems, whereas our approach decouples data access from the solving procedure; (3) it does not provide a declarative query language such as PaQL; (4) unlike SKETCHREFINE, Searchlight does not allow solvers to scale up to a very large number of variables.

Package queries are related to how-to queries [17], as they both use an ILP formulation to translate the original queries. However, there are several major differences between package queries and how-to queries: package queries specify tuple collections, whereas how-to queries specify updates to underlying datasets; package queries allow a tuple to appear multiple times in a package result, while how-to queries do not model repetitions; PaQL is SQL-based whereas how-to queries use a variant of Datalog; PaQL supports arbitrary Boolean formulas in the SUCH THAT clause, whereas how-to queries can only express conjunctive conditions.

The principal idea of constraint query languages (CQL) [15] is that a tuple can be generalized as a conjunction of constraints over variables. This general principle creates connections between declarative database languages and constraint programming. However, prior work focused on expressing constraints over tuple values, rather than over sets of tuples. PaQL follows a similar approach to CQL by embedding higher-order constraints in a declarative query language. However, our package query engine design allows for the direct use of ILP solvers as black box components, automatically transforming problems and solutions from one domain to the other. In contrast, CQL needs to appropriately adapt the algorithms themselves between the two domains, and existing literature does not provide this adaptation for the constraint types in PaQL.

There exists a large body of research in approximation algorithms for problems that can be modeled as integer linear programs. A typical approach is *linear programming relaxation* [24] in which the integrality constraints are dropped and variables are free to take on real values. These methods are usually coupled with *rounding* techniques that transform the real solutions to integer solutions with provable approximation bounds. None of these methods, however, can solve package queries on a large scale because they all assume that the LP solver is used on the entire problem. Another common approach to approximate a solution to an ILP problem is the *primal-dual method* [10]. All primal-dual algorithms, however, need to keep track of all primal and dual variables and the coefficient matrix, which means that none of these methods can be employed on large datasets. On the other hand, rounding techniques and primal-dual algorithms could potentially benefit from the SKETCHREFINE algorithm to break down their complexity on very large datasets.

Like package queries, *optimization under parametric aggregation constraints* (OPAC) queries [11] can construct sets of tuples that collectively satisfy summation constraints. However, existing solutions to OPAC queries have several shortcomings: (1) they do not handle tuple repetitions; (2) they only address *multi-attribute knapsack queries* – a subclass of package queries in which all global constraints are of the form $\mathsf{SUM}() \leq c$, with objective MAXIMIZE $\mathsf{SUM}()$; (3) they may return infeasible packages; (4) they require pre-computation of packages, which are then retrieved at query time using a multi-dimensional index. Package queries also encompass *submodular* optimization queries, whose recent approximate solutions use greedy distributed algorithms [18].

7. CONCLUSIONS

In this paper, we introduced a complete system that supports the declarative specification and efficient evaluation of package queries. We presented PaQL, a declarative extension to SQL, and we developed a flexible approximation method, with strong theoretical guarantees, for the evaluation of PaQL queries on large-scale datasets. Our experiments on real-world data demonstrate that our scalable evaluation strategy is effective and efficient over varied data sizes and queries.

Acknowledgements This material is based upon work supported by the National Science Foundation under grants IIS-1420941, IIS-1421322, and IIS-1453543.

8. REFERENCES

[1] S. Basu Roy, S. Amer-Yahia, A. Chawla, G. Das, and C. Yu. Constructing and exploring composite items. In *SIGMOD*, pages 843–854, 2010.

[2] A. Baykasoglu, T. Dereli, and S. Das. Project team selection using fuzzy optimization approach. *Cybernetic Systems*, 38(2):155–185, 2007.

[3] J. Bisschop. *AIMMS Optimization Modeling*. Paragon Decision Technology, 2006.

[4] M. Brucato, J. F. Beltran, A. Abouzied, and A. Meliou. Scalable package queries in relational database systems. *PVLDB*, 9(7):576–587, 2016.

[5] M. Brucato, R. Ramakrishna, A. Abouzied, and A. Meliou. PackageBuilder: From tuples to packages. *PVLDB*, 7(13):1593–1596, 2014.

[6] W. Cook and M. Hartmann. On the complexity of branch and cut methods for the traveling salesman problem. *Polyhedral Combinatorics*, 1:75–82, 1990.

[7] M. De Choudhury, M. Feldman, S. Amer-Yahia, N. Golbandi, R. Lempel, and C. Yu. Automatic construction of travel itineraries using social breadcrumbs. In *HyperText*, pages 35–44, 2010.

[8] T. Deng, W. Fan, and F. Geerts. On the complexity of package recommendation problems. In *PODS*, pages 261–272, 2012.

[9] R. A. Finkel and J. L. Bentley. Quad trees a data structure for retrieval on composite keys. *Acta informatica*, 4(1):1–9, 1974.

[10] M. X. Goemans and D. P. Williamson. The primal-dual method for approximation algorithms and its application to network design problems. *Approximation algorithms for NP-hard problems*, pages 144–191, 1997.

[11] S. Guha, D. Gunopulos, N. Koudas, D. Srivastava, and M. Vlachos. Efficient approximation of optimization queries under parametric aggregation constraints. In *VLDB*, pages 778–789, 2003.

[12] IBM CPLEX Optimization Studio. `http://www.ibm.com/software/commerce/optimization/cplex-optimizer/`.

[13] A. Kalinin, U. Çetintemel, and S. B. Zdonik. Interactive data exploration using semantic windows. In *SIGMOD*, pages 505–516, 2014.

[14] A. Kalinin, U. Çetintemel, and S. B. Zdonik. Searchlight: Enabling integrated search and exploration over large multidimensional data. *PVLDB*, 8(10):1094–1105, 2015.

[15] P. Kanellakis, G. Kuper, and P. Revesz. Constraint query languages. *Journal of Computer and System Sciences*, 1(51):26–52, 1995.

[16] T. Lappas, K. Liu, and E. Terzi. Finding a team of experts in social networks. In *SIGKDD*, pages 467–476, 2009.

[17] A. Meliou and D. Suciu. Tiresias: The database oracle for how-to queries. In *SIGMOD*, pages 337–348, 2012.

[18] B. Mirzasoleiman, A. Karbasi, R. Sarkar, and A. Krause. Distributed submodular maximization: Identifying representative elements in massive data. In *NIPS*, 2013.

[19] M. Padberg and G. Rinaldi. A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. *SIAM Review*, 33(1):60–100, 1991.

[20] A. G. Parameswaran, P. Venetis, and H. Garcia-Molina. Recommendation systems with complex constraints: A course recommendation perspective. *ACM TOIS*, 29(4):1–33, 2011.

[21] F. Pinel and L. R. Varshney. Computational creativity for culinary recipes. In *CHI*, pages 439–442, 2014.

[22] The Sloan Digital Sky Survey. `http://www.sdss.org/`.

[23] X. Wang, X. L. Dong, and A. Meliou. Data X-Ray: A diagnostic tool for data errors. In *SIGMOD*, pages 1231–1245, 2015.

[24] D. P. Williamson and D. B. Shmoys. *The design of approximation algorithms*. Cambridge University Press, 2011.

Technical Perspective: Optimized Wandering for Online Aggregation

Jeffrey F. Naughton
University of Wisconsin, Madison
naughton@cs.wisc.edu

There is a rich history in the DBMS research literature involving sampling to estimate the results of queries faster than they can be computed exactly. A particularly interesting example of this is "Online Aggregation" proposed by Hellerstein et al. in 1997 [2]. There the idea is to combine sampling with a creative and intuitive user interface. Briefly, when a query starts to run, Online Aggregation will quickly present an estimate of the result of the query (based on data sampled up to that point) and will also present a confidence interval around the estimate. As query execution continues, the estimate is refined, and the confidence interval shrinks.

Hidden in this attractive idea, however, are some difficult challenges. As an example, for queries that involve joins, the sampling process is in general slow, especially if most of the tuples from one relation participating in the join "match" with only a few tuples in the other relation. For 20 years the state of the art approach to this problem has been the "Ripple Join" [1]. The following paper by Li, Wu, Yi, and Zhao presents a highly effective alternative.

The main idea behind the wander join is to use the presence of indexes to speed the sampling, effectively making a random walk through the data join graph. The details of doing this efficiently (both computationally and statistically) are not obvious. The authors of this paper use a clever combination of sampling strategies from the statistical literature and an on-line optimization process to order the paths chosen for the random walk, in the process achieving much better computational and statistical properties than the previously state of the art algorithm. The authors convincingly prove this through experimentation with an open-source implementation in the Postgres database management system.

References

[1] P. J. Haas and J. M. Hellerstein. Ripple joins for online aggregation. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, SIGMOD '99, pages 287–298, New York, NY, USA, 1999. ACM.

[2] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, SIGMOD '97, pages 171–182, New York, NY, USA, 1997. ACM.

Wander Join and XDB: Online Aggregation via Random Walks

Feifei Li [1], Bin Wu [2], Ke Yi [2], Zhuoyue Zhao [1]
[1]University of Utah
[2]Hong Kong University of Science and Technology
{lifeifei, zyzhao}@cs.utah.edu {bwuac, yike}@cse.ust.hk

ABSTRACT

Joins are expensive, and online aggregation is an effective approach to explore the tradeoff between query efficiency and accuracy in a continuous, online fashion. However, the state-of-the-art approach, in both internal and external memory, is based on ripple join, which is still very expensive and needs strong assumptions (e.g., the tuples in a table are stored in random order). This paper proposes a new approach, the *wander join* algorithm, to the online aggregation problem by performing random walks over the underlying join graph. We also design an optimizer that chooses the optimal plan for conducting the random walks without having to collect any statistics *a priori*. Selection predicates and group-by clauses can be handled as well. We have developed an online engine called XDB by integrating wander join in the latest version of PostgreSQL. Extensive experiments using the TPC-H benchmark have shown the superior performance of wander join. The XDB implementation has demonstrated its practicality in a full-fledged database system.

1. INTRODUCTION

Joins are often considered to be the most central operation in relational databases, as well as the most costly one. For many of today's data-driven analytical tasks, users often need to pose ad hoc complex join queries involving multiple relational tables over gigabytes or even terabytes of data. The TPC-H benchmark, which is the industrial standard for decision-support data analytics, specifies 22 queries, 17 of which are joins, the most complex one involving 8 tables. For such complex join queries, even a leading commercial database system could take hours to process. This, unfortunately, is at odds with the low-latency requirement that users demand for interactive data analytics.

The research community has long realized the need for interactive data analysis and exploration, and in 1997, began a line of work known as "online aggregation" [7]. The observation is that such analytical queries do not really need a 100% accurate answer. It would be more desirable if the database could first quickly return an approximate answer with some form of quality guarantee (usually in the form of confidence intervals), while improving the accuracy as more time is spent. Then the user can stop the query processing as soon as the quality is acceptable.

Unfortunately, despite of many nice research results and well cited papers on this topic, online aggregation has had limited practical impact — we are not aware of any full-fledged, publicly available database system that supports it. Central to this line of work is the ripple join algorithm [5]. Its basic idea is to repeatedly take samples from each table, and only perform the join on the sampled tuples. The result is then scaled up to serve as an estimation of the whole join. However, the ripple join algorithm (including its many variants) has two critical weaknesses: (1) Its performance crucially depends on the fraction of the randomly selected tuples that actually join. However, we observe that this fraction is often exceedingly low, especially for equality joins (a.k.a. natural joins) involving multiple tables, while *all* queries in the TPC-H benchmark (thus arguably most joins used in practice) are natural joins. (2) It demands that the tuples in each table be stored in a random order.

This paper proposes a different approach, which we call *wander join*, to the online aggregation problem. Our basic idea is to not blindly take samples from each table and just hope that they join, but to make the process much more focused by leveraging indexes. Specifically, wander join takes a randomly sampled tuple only from one of the tables. After that, it conducts a random walk using indexes on the underlying join graph starting from that tuple. In every step of the random walk, only the "neighbors" of the already sampled tuples are considered, i.e., tuples in the unexplored tables that can actually join with them. Compared with the "blind search" of ripple join, this is more like a guided exploration, where we only look at portions of the data that can potentially lead to an actual join result. To summarize:

- We introduce *wander join* to achieve online aggregation for joins. The key idea is to model a join over k tables as a join graph, and then perform random walks in this graph. We show how the random walks lead to unbiased estimators for various aggregation functions, and give corresponding confidence interval formulas.
- It turns out that for the same join, there can be different ways to perform the random walks, which we call *walk plans*. We design an optimizer that chooses the optimal walk plan, without the need to collect any statistics of the data *a priori*.

- We have conducted extensive experiments to compare wander join with ripple join [5] and its system implementation DBO [2, 10]. The results show that wander join has outperformed ripple join and DBO by orders of magnitude in speed for achieving the same accuracy for in-memory data.
- We have implemented XDB by integrating wander join in PostgreSQL. On the TPC-H benchmark with tens of GBs of data, XDB is able to achieve 1% error with 95% confidence for most queries in a few seconds, whereas PostgreSQL may take minutes to return the exact results for the same queries.

2. BACKGROUND

Online aggregation. The concept of online aggregation was first proposed in the classic work by Hellerstein et al. [7]. The idea is to provide approximate answers with error guarantees (in the form of confidence intervals) continuously during the query execution process, where the approximation quality improves gradually over time. Rather than having a user wait for the exact answer, which may take an unknown amount of time, this allows the user to explore the efficiency-accuracy tradeoff, and to terminate the query execution whenever a good approximation quality is met.

For queries over one table, e.g., `SELECT SUM(quantity) FROM` R `WHERE discount` > 0.1, online aggregation is quite easy. The idea is to simply take samples from table R repeatedly, and compute the average of the sampled tuples (more precisely, on the value of the attribute on which the aggregation function is applied), which is then appropriately scaled up to get an unbiased estimator for the `SUM`. Standard statistical formulas can be used to estimate the confidence interval, which shrinks as more samples are taken [4].

Online aggregation for joins. For join queries, the problem becomes much harder. When we sample tuples from each table and join the sampled tuples, we get a sample of the join results. The sample mean can still serve as an unbiased estimator of the full join (after appropriate scaling), but these samples are *not* independently chosen from the full join results, even though the joining tuples are sampled from each table independently. Haas et al. [4,6] studied this problem in depth, and derived new formulas for computing the confidence intervals for such estimators, and later proposed the *ripple join* algorithm [5]. Ripple join repeatedly takes random samples from each table in a round-robin fashion, and keep all the sampled tuples in memory. Every time a new tuple is taken from one table, it is joined with all the tuples taken from other tables so far.

There have been many variants and extensions to the basic ripple join algorithm. First, if an index is available on one of the tables, say R_2, then for a randomly sampled tuple from R_1, we can find all the tuples in R_2 that join with it. Note that no random sampling is done on R_2. This variant is also known as *index ripple join*, which was actually noted before ripple join itself was invented [12, 13]. In general, for a multi-table join $R_1 \bowtie \cdots \bowtie R_k$, the index ripple join algorithm only does random sampling on one of the tables, say R_1. Then for each tuple t sampled from R_1, it computes $t \bowtie R_2 \bowtie \cdots \bowtie R_k$, and all the joined results are returned as samples from the full join.

Problem formulation. The type of queries we aim to support is a SQL query of the form

`SELECT g, AGG(expression) FROM` $R_1, R_2, \ldots, R_k$
`WHERE join conditions AND selection predicates GROUP BY g`

where `AGG` can be any of the standard aggregation functions such as `SUM`, `AVE`, `COUNT`, `VARIANCE`, and `expression` can involve any attributes of the tables. The `join conditions` consist of equality or inequality conditions between pairs of the tables, and `selection predicates` can also be applied to any number of the tables. For example, in the following query, the first three terms in the `WHERE` clause are join conditions while the others are selection predicates:

`SELECT SUM(l_extended_price` $*$ `(1 −` `l_discount))`
`FROM` $nation, customer, orders, lineitem$
`WHERE n_nationkey = c_nationkey AND c_custkey = o_custkey`
`AND o_orderkey = l_orderkey AND n_name =′ US′ AND l_flag =′ R′`

At any point in time during query processing, the algorithm should output an estimator $\tilde{Y}$ for `AGG(expression)` together with a confidence interval, i.e.,

$$\Pr[|\tilde{Y} - \texttt{AGG(expression)}| \leq \varepsilon] \geq \alpha.$$

Here, ε is called the *half-width* of the confidence interval and α the *confidence level*. The user should specify one of them and the algorithm will continuously update the other as time goes on. The user can terminate the query when it reaches the desired level. Alternatively, the user may also specify a time limit on the query processing, and the algorithm should return the best estimate obtainable within the limit, together with a confidence interval.

3. WANDER JOIN

3.1 Wander join on a simple example

For concreteness, we first illustrate how wander join works on the natural join between 3 tables R_1, R_2, R_3:

$$R_1(A, B) \bowtie R_2(B, C) \bowtie R_3(C, D), \tag{1}$$

where $R_1(A, B)$ means that R_1 has two attributes A and B, etc. The natural join returns all combinations of tuples from the 3 tables that have matching values on their common attributes. We assume that R_2 has an index on attribute B, R_3 has an index on attribute C, and the aggregation function is `SUM`(D).

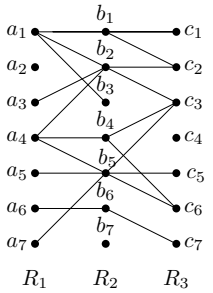


Figure 1: The 3-table join data graph: there is an edge between two tuples if they can join. Note that this represents a join query with general join conditions that are not necessarily natural/equi-join.

We model the join relationships among the tuples as a graph. More precisely, each tuple is modeled as a vertex and there is an edge between two tuples if they can join.

For this natural join, it means that the two tuples have the same value on their common attribute. We call the resulting graph the *join data graph* (this is to be contrasted with the *join query graph* introduced later). For example, the join data graph for the 3-table natural join (1) may look like the one in Figure 1. This way, each join result becomes a path from some vertex in R_1 to some vertex in R_3, and sampling from the join boils down to sampling a path. Note that this graph is completely *conceptual*: we do not need to actually construct the graph to do path sampling.

A path can be randomly sampled by first picking a vertex in R_1 uniformly at random, and then "randomly walking" towards R_3. Specifically, in every step of the random walk, if the current vertex has d neighbors in the next table (which can be found efficiently by the index), we pick one uniformly at random to walk to.

One problem an acute reader would immediately notice is that, different paths may have different probabilities. In the example above, the path $a_1 \to b_1 \to c_1$ has probability $\frac{1}{7}\cdot\frac{1}{3}\cdot\frac{1}{2}$, while $a_6 \to b_6 \to c_7$ has probability $\frac{1}{7}\cdot 1\cdot 1$. If the value of the D attribute on c_7 is very large, then obviously this would tilt the balance, leading to an overestimate. Ideally, each path should be sampled with equal probability so as to ensure unbiasedness. However, it is well known that random walks in general do not yield a uniform distribution.

Fortunately, a technique known in the statistics literature as the *Horvitz-Thompson estimator* [8] can be used to remove the bias easily. Suppose path γ is sampled with probability $p(\gamma)$, and the expression on γ to be aggregated is $v(\gamma)$, then $v(\gamma)/p(\gamma)$ is an unbiased estimator of $\sum_\gamma v(\gamma)$, which is exactly the `SUM` aggregate we aim to estimate. This can be easily proved by the definition of expectation, and is also very intuitive: We just penalize the paths that are sampled with higher probability proportionally. Also note that $p(\gamma)$ can be computed easily on-the-fly as the path is sampled. Suppose $\gamma = (t_1, t_2, t_3)$, where t_i is the tuple sampled from R_i, then we have

$$p(\gamma) = \frac{1}{|R_1|} \cdot \frac{1}{d_2(t_1)} \cdot \frac{1}{d_3(t_2)}, \text{ where} \tag{2}$$

$d_{i+1}(t_i)$ is the number of tuples in R_{i+1} that join with t_i.

Finally, we independently perform multiple random walks, and take the average of the estimators $v(\gamma_i)/p_i$. Since each $v(\gamma_i)/p_i$ is an unbiased estimator of the `SUM`, their average is still unbiased, and the variance of the estimator reduces as more paths are collected.

A subtle question is what to do when the random walk gets stuck, for example, when we reach vertex b_3 in Figure 1. In this case, we should not reject the sample, but return 0 as the estimate, which will be averaged together with all the successful random walks. This is because even though this is a failed random walk, it is still in the probability space. It should be treated as a value of 0 for the Horvitz-Thompson estimator to remain unbiased. Too many failed random walks will slow down the convergence of estimation, and we will deal with the issue in Section 4.

3.2 Wander join for acyclic queries

Although the algorithm above is described on a simple 3-table *chain join*, it can be extended to arbitrary joins easily. In general, we consider the *join query graph* (or *query graph* in short), where each table is modeled as a vertex, and there is an edge between two tables if there is a join condition between the two; see Figure 2 for examples.

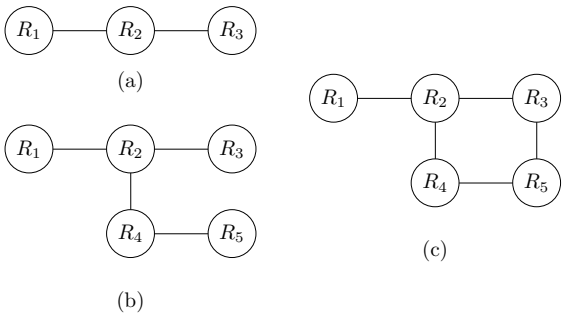


Figure 2: The join query graph for a (a) chain join; (b) acyclic join; (c) cyclic join.

When the join query graph is acyclic, wander join can be extended in a straightforward way. First, we need to fix a *walk order* such that each table in the walk order must be adjacent (in the query graph) to another one earlier in the order. For example, for the query graph in Figure 2(b), R_1, R_2, R_3, R_4, R_5 and R_2, R_3, R_4, R_5, R_1 are both valid walk orders, but R_1, R_3, R_4, R_5, R_2 is not since R_3 (resp. R_4) is not adjacent to R_1 (resp. R_1 or R_3) in the query graph. (Different walk orders may lead to very different performance, and we will discuss how to choose the best one in Section 4.)

Next, we simply perform the random walks as before, following the given order. The only difference is that a random walk may now consist of both "walks" and "jumps". For example, using the order R_1, R_2, R_3, R_4, R_5 on Figure 2(b), after we have reached a tuple in R_3, the next table to walk to is R_4, which is connected to the part already walked via R_2. So we need to jump back to the tuple we picked in R_2, and continue the random walk from there.

Finally, we need to generalize Equation (2). Let $d_j(t)$ be the number of tuples in R_j that can join with t, where t is a tuple from another table that has a join condition with R_j. Suppose the walk order is $R_{\lambda(1)}, R_{\lambda(2)}, \ldots, R_{\lambda(k)}$, and let $R_{\eta(i)}$ be the table adjacent to $R_{\lambda(i)}$ in the query graph but appearing earlier in the order. Note that for an acyclic query graph and a valid walk order, $R_{\eta(i)}$ is uniquely defined. Then for the path $\gamma = (t_{\lambda(1)}, \ldots, t_{\lambda(k)})$, where $t_{\lambda(i)} \in R_{\lambda(i)}$, the sampling probability of the path γ is

$$p(\gamma) = \frac{1}{|R_{\lambda(1)}|} \prod_{i=2}^{k} \frac{1}{d_{\lambda(i)}(t_{\eta(i)})}. \tag{3}$$

3.3 Wander join for cyclic queries

The algorithm for acyclic queries can also be extended to handle query graphs with cycles. Given a cyclic query graph, e.g., the one in Figure 2(c), we first find any spanning tree, such as the one in Figure 2(b). Then we just perform the random walks on this spanning tree as before. After we have sampled a path γ on the spanning tree, we need to put back the non-spanning tree edges, e.g., (R_3, R_5), and check that γ satisfies the join conditions on these edges. For example, after we have sampled a path $\gamma = (t_1, t_2, t_3, t_4, t_5)$ on Figure 2(b) (assuming the walk order R_1, R_2, R_3, R_4, R_5), then we need to verify that γ satisfies the non-spanning tree edge (R_3, R_5), i.e., t_3 should join with t_5. If they do not join, we consider γ as a failed random walk and return an estimator with value 0.

3.4 Estimators and confidence intervals

To derive estimators and confidence interval formulas for various aggregation functions, we establish an equivalence between wander join and sampling from a single table with selection predicates, which has been studied by Haas [4]. Imagine that we have a single table that stores all the paths in the join data graph, including the full paths, as well as the partial paths (like $a_1 \to b_3$). Wander join essentially samples from this imaginary table, though non-uniformly.

Suppose we have performed a total of n random walks $\gamma_1, \ldots, \gamma_n$. For each γ_i, let $v(i)$ be the value of the expression on γ_i to be aggregated, and set $u(i) = 1/p(\gamma_i)$ if γ_i is a successful walk, and 0 otherwise. With this definition of u and v, we can rewrite the estimator for `SUM` as $\frac{1}{n}\sum_{i=1}^{n} u(i)v(i)$. We observe that this has exactly the same form as the one in [4] for estimating the `SUM` for a single table with a selection predicate, except for two differences: (1) in [4], $u(i)$ is set to 1 if γ_i satisfies the selection predicate and 0 otherwise; and (2) [4] does uniform sampling over the table, while our sampling is non-uniform. However, by going through the analysis in [4], we realize that it holds for any definition of u and v, and for any sampling distribution. Thus, all the results in [4] carry over to our case, but with u and v defined in our way. The detailed formulas can be found in [11]; all of them can be computed easily in $O(n)$ time.

3.5 Selection predicates and Group By Clause

Wander join can deal with arbitrary selection predicates in the query easily: in the random walk process, whenever we reach a tuple t for which there is a selection predicate, we check if it satisfies the predicate, and fail the random walk immediately if not.

If the starting table of the random walk has an index on the attribute with a selection predicate, and the predicate is an equality or range condition, then we can directly sample a tuple that satisfies the condition from the index, using Olken's method [14]. Correspondingly, we replace $|R_{\lambda(1)}|$ in (3) by the number of tuples in $R_{\lambda(1)}$ that satisfy the condition, which can also be computed from the index. This removes the impact of the predicate on the performance of the random walk, thus it is preferable to start from such a table. More discussion will be devoted on this topic under walk plan optimization in Section 4.

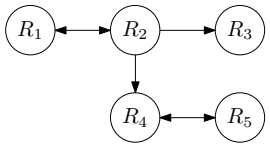
Wander join supports a Group By clause by maintaining multiple estimators simultaneously during the random walk process, one per group with respect to the grouping attribute(s). Each random walk is pushed to the group it belongs to and used to update the corresponding estimator.

4. WALK PLAN OPTIMIZER

Different orders in which to perform the random walk may lead to very different performance. This is akin to choosing the best physical plan for executing a query. So we term different ways to perform the random walks as *walk plans*. A relational database optimizer usually needs statistics to be collected from the tables *a priori*, so as to estimate various intermediate result sizes for multi-table join optimization. In this section we present a walk plan optimizer that chooses the best walk plan without the need to collect statistics.

4.1 Walk plan generation

We first generate all possible walk plans. Recall that the constraint we have for a valid walk order is that for each table R_i (except the first one), there must exist a table R_j earlier in the order such that there is a join condition between R_i and R_j. In addition, R_i should have an index on the attribute that appears in the join condition. Note that the join condition does not have to be equality. It can be for instance an inequality or even a range condition, such as $R_j.A \le R_i.B \le R_j.A + 100$, as long as R_i has an index on B that supports range queries (e.g., a B-tree).



R_1, R_2, R_3, R_4, R_5
R_1, R_2, R_4, R_3, R_5
R_1, R_2, R_4, R_5, R_3
R_2, R_1, R_3, R_4, R_5
R_2, R_1, R_4, R_3, R_5
R_2, R_1, R_4, R_5, R_3
R_2, R_3, R_1, R_4, R_5
R_2, R_3, R_4, R_1, R_5
R_2, R_3, R_4, R_5, R_1
R_2, R_4, R_1, R_3, R_5
R_2, R_4, R_1, R_5, R_3
R_2, R_4, R_3, R_1, R_5
R_2, R_4, R_3, R_5, R_1
R_2, R_4, R_5, R_1, R_3
R_2, R_4, R_5, R_3, R_1

Figure 3: A directed join query graph and all its walk plans.

To generate all possible walk orders, we first add directions to each edge in the join query graph. Specifically, for an edge between R_i and R_j, if R_i has an index on its attribute in the join condition, we have a directed edge from R_j to R_i; similarly if R_j has an index on its attribute in the join condition, we have a directed edge from R_i to R_j. For example, after adding directions, the query graph in Figure 2(b) might look like the one in Figure 3, and all possible walk plans are listed on the side. These plans can be enumerated by a simple backtracking algorithm. Note that there can be exponentially (in the number of tables) many walk plans. However, this is not a real concern because (1) there cannot be too many tables, and (2) more importantly, having many walk plans does not have a major impact on the plan optimizer, which we shall see later.

We can similarly generate all possible walk plans for cyclic queries, just that some edges will not be walked, and they will have to be checked after the random walk, as described in Section 3.3. We call them *non-tree* edges, since the part of the graph that is covered by the random walk form a tree. An example is given in Figure 4.

4.2 Walk plan optimization

The performance of a walk order depends on many factors. However, we observe that ultimately, the performance of the random walk is measured by the variance of the final estimator after a given amount of time, say t. Let X_i be the estimator from the i-th random walk (e.g., $u(i)v(i)$ for `SUM` if the walk is successful and 0 otherwise), and let T be the running time of one random walk, successful or not. Suppose a total of W random walks have been performed within time t. Then the final estimator is $\frac{1}{W}\sum_{i=1}^{W} X_i$. We show that

$$\mathsf{Var}\left[\frac{1}{W}\sum_{i=1}^{W} X_i\right] = \mathsf{Var}[X_1]\mathsf{E}[T]/t.$$

Thus, for a given amount of time t, the variance of the final estimator is proportional to $\mathsf{Var}[X_1]\mathsf{E}[T]$. The next observation is that both $\mathsf{Var}[X_1]$ and $\mathsf{E}[T]$ can also be estimated

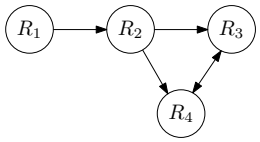


Figure 4: Walk plan for a cyclic query graph.

by the random walks themselves! In particular, $\mathsf{Var}[X_1]$ is just estimated as another aggregation function; for $\mathsf{E}[T]$, we simply count the number of index entries looked up, or the number of I/Os in external memory, in each random walk, and take the average.

Now, for each walk order, we perform a certain number of "trial" random walks and estimate $\mathsf{Var}[X_1]$ and $\mathsf{E}[T]$. Then we compute the product $\mathsf{Var}[X_1]\mathsf{E}[T]$ and pick the order with the minimum $\mathsf{Var}[X_1]\mathsf{E}[T]$. How to choose the number of trials is the classical sample size determination problem [1], which again depends on many factors such as the actual data distribution, the level of precision required, etc. We adopt the following strategy: We conduct random walks following each plan in a round-robin fashion, and stop until at least one plan has accumulated at least τ successful walks. Then we pick the plan with the minimum $\mathsf{Var}[X_1]\mathsf{E}[T]$ that has at least $\tau/2$ successful walks. This is actually motivated by association rule mining, where a rule must both be good and have a minimum support level. In our implementation, we use a default threshold of $\tau = 100$.

Finally, we observe that all the trial runs are not wasted. Since each random walk, no matter which plan it follows, returns an unbiased estimator. So we can include all the random walks, before and after the optimal one has been picked, in computing the final estimator. The confidence interval is also computed with all these random walks. This is unlike traditional query optimization, where the cost incurred by the optimizer itself is pure "overhead".

5. XDB: INTEGRATING WANDER JOIN INSIDE A DBMS ENGINE

Wander Join can be easily integrated into existing database engines. To demonstrate this point, we have developed XDB (approXimate DB) by integrating wander join in the latest version of PostgreSQL (version 9.4; in particular, 9.4.2). Our implementation covers the entire pipeline from SQL parsing to plan optimization to physical execution. We build secondary B-tree indexes on all the join attributes and the attributes used in the selection predicates. XDB is now open-sourced at `https://github.com/initialDLab/xdb`.

XDB extends PostgreSQL's parser, query optimizer, and query executor to support keywords like `CONFIDENCE`, `ONLINE`, `WITHINTIME`, and `REPORTINTERVAL`. We also integrated the plan optimizer of wander join into the query optimizer of PostgreSQL. For example, an example based on Q3 of TPC-H benchmark is:

```
SELECT ONLINE
SUM(l_extendedprice * (1 - l_discount)), COUNT(*)
FROM customer, orders, lineitem
WHERE c_mktsegment='BUILDING' AND c_custkey=o_custkey
AND l_orderkey=o_orderkey
WITHINTIME 20000 CONFIDENCE 95 REPORTINTERVAL 1000
```

This tells the engine that it is an online aggregation query, such that the engine should report the estimations and their associated confidence intervals, calculated with respect to 95% confidence level, for both `SUM` and `COUNT` every 1000 milliseconds for up to 20000 milliseconds.

Online aggregation queries are passed to an optimizer specific to wander join. The optimizer builds the join query graph and generates valid walk paths from the join query graph. The optimizer also replaces aggregation operators with online aggregation estimators and relative confidence interval operators. If the query contains an `INITSAMPLE` clause, which allows the engine to execute a number of trial runs using multiple paths to find the best walk order, all the valid walk paths are retained in the query plan. The query executor later iterates through all the walk paths, performs a number of trial runs as specified by the query and computes a rejection rate estimation and a variance estimation. It then orders the walk plans by the rejection rate and breaks tie (rejection rates differed within 5%) by the variance estimation.

The executor extracts samples from primary or secondary B-tree indexes one by one given a walk path. The B-tree indexes are augmented with counts of subtrees in their internal nodes. The executor uses the counts to find the degrees of the tuples in the join data graph and extract samples. Selection predicates are immediately applied when the related tuples are sampled, instead of waiting until the walk is complete. Once a walk completes, the executor maintains a few aggregations of the samples and probabilities for the estimators. The executor returns the current estimators and relative confidence intervals periodically. Finally, it returns an empty tuple when the time budget is used up, which informs PostgreSQL that no more tuples are available.

A Zeppelin frontend was also developed as part of the XDB system, where its visualization module was modified so that an online visualization of the (continuously updated) query results as well as the confidence intervals is enabled.

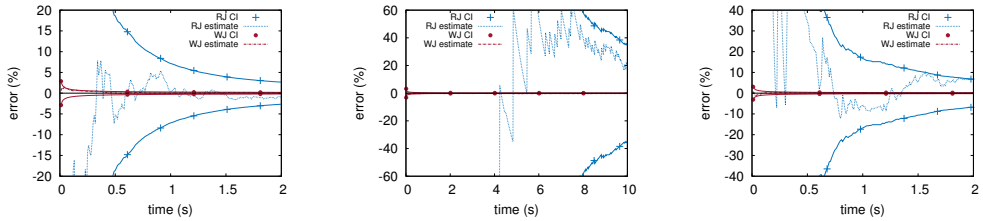
The only system implementation available for ripple join is the DBO system [2, 9, 10]. In fact, the algorithm implemented in DBO is much more complex than the basic ripple join in order to deal with limited memory, as described in these papers. We compared XDB with Turbo DBO, using the code at `http://faculty.ucmerced.edu/frusu/Projects/DBO/dbo.html`, as a system-to-system comparison. Note that due to the random order storage requirement, DBO was built from ground up. Currently it is still a prototype that supports online aggregation only (i.e., no support for other major features in a RDBMS engine, such as transactions, locking, etc.). On the other hand, XDB retains the full functionality of a RDBMS, with online aggregation as an added feature. Thus, this comparison can only be to our disadvantage due to the system overhead inside a full-fledged DBMS for supporting other features and functionality.

Note that the original DBO papers [9] compared the DBO engine against the PostgreSQL database by running the same queries in both systems. We did exactly the same in our experiments, but using XDB (which is a PostgreSQL with wander join implemented inside its kernel).

6. EXPERIMENTS

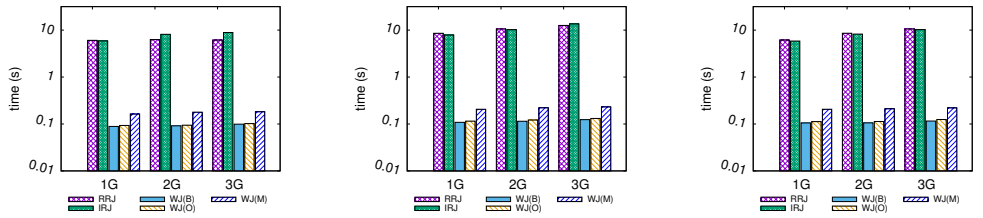
6.1 Experimental setup

We have evaluated the performance of wander join in comparison with ripple join and its variants, the DBO engine, under two settings: using a standalone implementation and a system implementation (XDB) respectively. In the stan-



(a) Q3 (b) Q7 (c) Q10

Figure 5: Standalone implementation: Confidence intervals and estimates on barebone queries on 2GB TPC-H data set; confidence level is 95%.



(a) Q3 (b) Q7 (c) Q10

Figure 6: Standalone implementation: Time to reach ±1% confidence interval and 95% confidence level on TPC-H data sets of different sizes.

dalone setting, we implemented both wander join and ripple join in C++. We ensure that the index structures fit in memory; in fact, all the indexes together take space that is a very small fraction of the total amount of data, because they are all secondary indexes, storing only pointers to the actual data records, which have many other attributes that are not indexed. Building these indexes is very efficient; in fact, they can be built with minimal overhead while loading data from the file system to the memory (which one has to do anyway). Similarly, for ripple join, we gave it enough memory so that all samples taken can be kept in memory.

The standalone implementation gives an ideal environment to both algorithms without any system overhead.

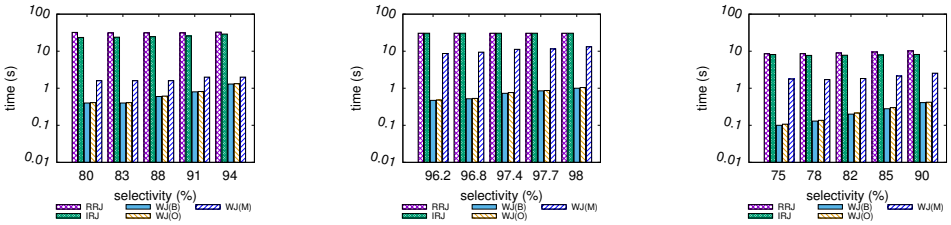
Data and queries. We used the TPC-H benchmark data and queries for the experiments, which were also used by the DBO work [2, 9, 10]. We used 5 tables, `nation`, `supplier`, `customer`, `orders`, and `lineitem`. We used the TPC-H data generator with the appropriate scaling factor to generate data sets of various sizes. We picked queries Q3 (3 tables), Q7 (6 tables; the `nation` table appears twice in the query) and Q10 (5 tables) in the TPC-H specification for testing.

Effect of data skewness. There are two types of skewness. Degree skewness refers to the skewness in the distribution of the number of tuples in one table that join another, while value skewness is the skewness of the distribution of the values being aggregated. The degree skewness will negatively impact the random walk process of wander join if a good walk order is not selected. This issue is addressed by our walk order optimization. Depending on how the cardinality of the join changes, it usually has no impact or even positive impact on the efficiency of wander join. In contrast, degree skewness often leads to worse performance for ripple join due to the increasing join sparsity for most tuples. On the other hand, value skewness has a negative impact on all online aggregation methods because the higher variance of aggregated values leads to a larger variance of the estimator. Unless prior knowledge of the value distribution is available, the effectiveness of (any) sampling methods will be affected.

6.2 Results on standalone implementation

We first run wander join (WJ) and ripple join (RJ) on a 2GB data set, i.e., the entire TPC-H database is 2GB, using the "barebone" joins of Q3, Q7, and Q10, where we drop all the selection predicates. Figure 5 plots how the *confidence interval* (CI) shrinks over time, with the confidence level set at 95%, as well as the *estimates* returned by the algorithms. They are shown as a percentage error compared with the true answer (obtained offline by running the exact joins to full completion). We can see that WJ converges much faster than RJ, due to the much more focused exploration strategy. Meanwhile, the estimates returned are indeed within the confidence interval almost all the time. For example, wander join converges to 1% confidence interval in less than 0.1 second whereas ripple join takes more than 4 seconds to reach 1% confidence interval. The full exact join on Q3, Q7, and Q10 in this case is 18 seconds, 28 seconds, and 19 seconds, respectively, using hash join.

Next, we ran the same queries on data sets of varying sizes. Now we include both the random order ripple join (RRJ) and the index-assisted ripple join (IRJ). For wander join, we also considered two other versions to see how the plan optimizer worked. WJ(B) is the version where the optimal plan is used (i.e., we run the algorithm with every plan and report the best result); WJ(M) is the version where we use the median plan (i.e., we run all plans and report the median result). WJ(O) is the version where we use the optimizer to automatically choose the plan, and the time spent by the optimizer is included. In Figure 6 we report the time spent by each algorithm to reach ±1% confidence interval with 95% confidence level on data sets of sizes 1GB, 2GB, and 3GB. We also report the time costs of the optimizer in Table 1. From the results, we can draw the following observations: (1) Wander join is in general faster than ripple join by two orders of magnitude to reach the same confidence interval. (2) The running time of ripple join increases with N, the data size, though mildly. (3) The running time of wander join is not affected by N. This also agrees with our analysis: When hash tables are used, its efficiency is independent of N altogether. (4) The optimizer has very low overhead, and is

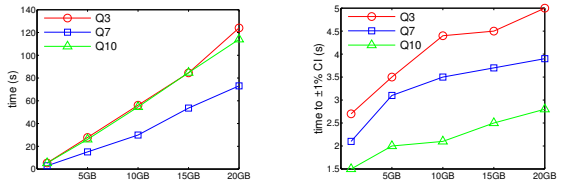


(a) Q3 (b) Q7 (c) Q10

Figure 7: Standalone implementation: Time to reach ±1% confidence interval and 95% confidence level on the 2GB TPC-H data set with multiple selection predicate of varying selectivity.

	size (GB)	optimization (ms)	execution (ms)
Q3	1	2.8	88.7
	2	2.8	91.3
	3	2.9	101.9
Q7	1	6.4	106.1
	2	6.4	112.1
	3	6.6	123.7
Q10	1	7.0	105
	2	7.3	105.6
	3	8.8	116

Table 1: Standalone implementation: Time cost of walk plan optimization (execution time to reach ±1% confidence interval and 95% confidence level on TPC-H data sets of different sizes).



(a) PostgreSQL full join (b) Wander join in PG

Figure 8: XDB: system implementation experimental results with sufficient memory – 32GB memory.

very effective. In fact, from the figures, we see that WJ(B) and WJ(O) have almost the same running time, meaning that the optimizer spends almost no time and indeed has found either the best plan or a very good plan that is almost as good as the best plan. Recall that all the trial runs used in the optimizer for selecting a good plan are not wasted; they also contribute to building the estimators. For barebone queries, many plans actually have similar performance, as seen by the running time of WJ(M), so even the trial runs are of good quality.

Finally, we put back the selection predicates to the queries. Figure 7 shows the time to reach ±1% confidence interval with 95% confidence level for the algorithms on the 2GB data set with all the predicates are put back. Here, we measure the overall selectivity of all the predicates as:

$$1 - (\text{join size with predicates})/(\text{barebone join size}), \quad (4)$$

so higher means more selective.

From the results, we see that one selection predicate has little impact on the performance of wander join, because most likely its optimizer will elect to start the walk from that table. Multiple highly selective predicates do affect the performance of wander join, but even in the worst case, wander join maintains a gap with ripple join of more than an order of magnitude.

These experiments also demonstrate the importance of the plan optimizer: With multiple highly selective predicates, a mediocre plan can be much worse than the optimal one, and the plan optimizer almost always picks the optimal or a close-to-optimal plan with nearly no overhead. Note that in this case we do have poor plans, so some trial random walks may contribute little to the estimation. However, the good plans can accumulate $\tau = 100$ successful random walks very quickly, so we do not waste too much time anyway.

6.3 Results on system implementation

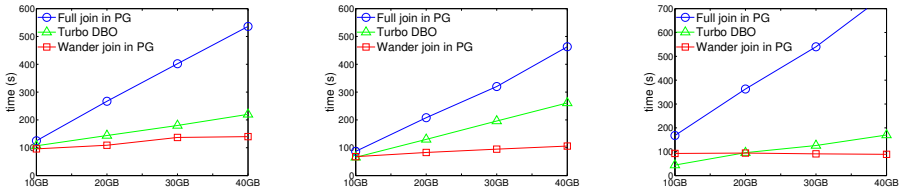
For the experimental evaluation on XDB, which is our PostgreSQL integration and implementation of wander join, we first tested how it performs when there is sufficient memory, and then tested the case when memory is severely limited. We compared against Turbo DBO in the latter case. Turbo DBO [2] is an improvement to the original DBO engine, that extends ripple join to data on external memory with many optimizations.

When there is sufficient memory. Due to the low-latency requirement for data analytical tasks and thanks to growing memory sizes, database systems are moving towards the "in-memory" computing paradigm. So we first would like to see how our system performs when there is sufficient memory. For this purpose, we used a machine with 32GB memory and data sets of sizes up to 20GB. We ran both online version of XDB and the built-in PostgreSQL full join in XDB on the same queries, both through the standard PostgreSQL SQL query interface.

Note that since we have built indexes on all the join attributes and there is sufficient memory, the PostgreSQL optimizer chose index join for all the join operators. We used Q3, Q7, and Q10 with all the selection predicates.

The results in Figure 8 clearly indicate a linear growth of the full join, which is as expected because the index join algorithm has running time linear in the table size. Also because all joins are primary key-foreign key joins, the intermediate results have roughly linear size. On the other hand, the data size has a mild impact on the performance of wander join. For example, the time to reach ±1% confidence interval for Q7 merely increases from 3 seconds to 4 seconds, when the data size increases from 5GB to 20GB in Figure 8(b). By our analysis and the internal memory experimental results, the total number of random walk steps should be independent of the data size. However, because we use B-tree indexes, whose access cost grows logarithmically as data gets larger, the cost per random walk step might grow slightly. Nevertheless, PostgreSQL with wander join reaching 1% CI has outperformed the PostgreSQL with full join by more than one order of magnitude when data size grows.

We have also run Turbo DBO in this case. However, it turned out that Turbo DBO spends even more time than PostgreSQL's full join, so we do not show its results. This



(a) Q3 (b) Q7 (c) Q10

Figure 9: XDB: system implementation experimental results with limited memory – 4GB memory.

seems to contradict with the results in [10]. In fact, this is because DBO is intentionally designed for large data and small memory. In the experiments of [10], the machine used had only 2GB of memory. With such a small memory, PostgreSQL had to resort to sort-merge join or nested-loop join for each join operator, which is much less efficient than index join (for in-memory data). Meanwhile, DBO follows the framework of sort-merge join, so it is actually not surprising that it is not as good as index joins for in-memory data. In our next set of experiments where we limit the memory size, we do see that DBO performs better than the full join.

When memory is limited. In our last set of experiments, we used a machine with only 4GB memory, and ran the same set of experiments as above on data sets of sizes starting from 10GB and increasing to 40GB. The time for wander join inside PostgreSQL and Turbo DBO to reach ±5% confidence interval with 95% confidence level, as well as the time of the full join in PostgreSQL, are shown in Figure 9.

From the results, we see that a small memory has a significant impact on the performance of wander join. The running time increases from a few seconds in Figure 8 to more than 100 seconds in Figure 9, and that's after we have relaxed the target confidence interval from ±1% to ±5%. The reason is obviously due to the random access nature of the random walks, which now has a high cost due to excessive page swapping. Nevertheless, this is a "one-time" cost, in the sense that each random walk step is now much more expensive, but the number of steps is still not affected. After the one-time, sudden increase when data size exceeds main memory, the total cost remains almost flat afterward. In other words, the cost of wander join in this case is still independent of the data size, albeit to a small increase in the index accessing cost (which grows logarithmically with the data size if B-tree is used). Hence, wander join still enjoys excellent scalability as data size continues to grow.

On the other hand, both the full join and DBO clearly have a linear dependency on the data size, though at different rates. On the 10GB and 20GB data sets, wander join and DBO have similar performance, but eventually wander join would stand out on very large data sets.

Anyway, spending 100 seconds just to get a ±5% estimate does not really meet the requirement of interactive data analytics, so strictly speaking both wander join and DBO have failed in this case (when data has significantly exceeded the memory size). Fundamentally, online aggregation requires some form of randomness so as to have a statistically meaningful estimation, which is at odds with the sequential access nature of hard disks. This appears to be an inherent barrier for this line of work. However, as memory sizes grow larger and memory clouds get more popular (for example, using systems like RAMCloud [15] and FaRM [3]), with the SSDs as an additional storage layer, in the end we may not have to deal with this barrier at all.

7. CONCLUSION

We have open sourced the XDB engine at `https://github.com/InitialDLab/XDB`. In addition to the integration with the PostgreSQL kernel, we have also designed and implemented a front-end interface using Apache Zeppelin, which is able to show the query results in the form of table, line plot and other visualization representation in a continuous online fashion. For future work, an important open problem is to extend online aggregations to nested queries.

8. ACKNOWLEDGMENTS

Feifei Li and Zhuoyue Zhao are supported in part by NSF grants 1251019, 1302663, 1443046 and NSFC grant 61428204. Bin Wu and Ke Yi are supported by HKRGC under grants GRF-621413, GRF-16211614, and GRF-16200415. The authors greatly appreciate the valuable feedback provided by the anonymous SIGMOD reviewers and Professor Jeffrey Naughton in preparing this manuscript.

9. REFERENCES

[1] G. Casella and R. L. Berger. *Statistical Inference*. Duxbury Press, 2001.

[2] A. Dobra, C. Jermaine, F. Rusu, and F. Xu. Turbo charging estimate convergence in dbo. In *VLDB*, 2009.

[3] A. Dragojevic, D. Narayanan, M. Castro, and O. Hodson. FaRM: Fast remote memory. In *NSDI*, pages 401–414, 2014.

[4] P. J. Haas. Large-sample and deterministic confidence intervals for online aggregation. In *SSDBM*, 1997.

[5] P. J. Haas and J. M. Hellerstein. Ripple joins for online aggregation. In *SIGMOD*, pages 287–298, 1999.

[6] P. J. Haas, J. F. Naughton, S. Seshadri, and A. N. Swami. Selectivity and cost estimation for joins based on random sampling. *JCSS*, 52:550–569, 1996.

[7] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *SIGMOD*, 1997.

[8] D. G. Horvitz and D. J. Thompson. A generalization of sampling without replacement from a finite universe. *JASA*, 47:663–685, 1952.

[9] C. Jermaine, S. Arumugam, A. Pol, and A. Dobra. Scalable approximate query processing with the DBO engine. In *SIGMOD*, 2007.

[10] C. Jermaine, S. Arumugam, A. Pol, and A. Dobra. Scalable approximate query processing with the DBO engine. *ACM TODS*, 33(4), Article 23, 2008.

[11] F. Li, B. Wu, K. Yi, and Z. Zhao. Wander join: Online aggregation via random walks. In *SIGMOD*, 2016.

[12] R. J. Lipton and J. F. Naughton. Query size estimation by adaptive sampling. In *PODS*, 1990.

[13] R. J. Lipton, J. F. Naughton, and D. A. Schneider. Practical selectivity estimation through adaptive sampling. In *SIGMOD*, 1990.

[14] F. Olken. *Random Sampling from Databases*. PhD thesis, University of California at Berkeley, 1993.

[15] J. K. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, D. Ongaro, G. M. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman. The case for RAMCloud. *Commun. ACM*, 54(7):121–130, 2011.

Technical Perspective: Scaling Machine Learning via Compressed Linear Algebra

Zachary G. Ives
University of Pennsylvania
zives@cis.upenn.edu

Demand for more powerful "big data analytics" solutions has spurred a great deal of interest in the core programming models, abstractions, and platforms for next-generation systems. For these problems, a complete solution would address data wrangling and processing, and support analytics over data of any modality or scale. It would support a wide array of machine learning algorithms, but also provide primitives for building new ones. It should be customizable, scale to vast volumes of data, and map to modern multicore, GPU, co-processor, and compute cluster hardware.

In pursuit of these goals, novel techniques and solutions are being developed by machine learning researchers (e.g., high-performance libraries like Theano [6], runtime systems like GraphLab [5]), in the database and distributed systems research communities (e.g., distributed data analytics engines like Spark [7] and Flink [3]), and in industry by major technology players (e.g., Google's TensorFlow [1] and IBM/Apache's SystemML [4]). These libraries and platforms support multiple development languages, provide abstract datatypes for machine learning over data, and include compilers and runtime systems optimized for distributed execution on modern hardware.

The database community excels in developing techniques for cost-estimating and optimizing declarative programs, and in exploiting *data independence* to optimize data placement and layout for performance. Elgohary et al's work on "Scaling Machine Learning via Compressed Linear Algebra," which appeared in the Proceedings of the VLDB Endowment [2], was conducted within IBM and Apache's SystemML declarative machine learning project. It shows just how effective such database techniques can be in a machine learning setting. The authors observe that the core data objects in machine learning – feature matrices, weight vectors – tend to have repeated values as well as regular structure, and may be quite large. Machine learning tasks over such data are composed from lower-level linear algebra operations. Such operations generally involve repeated floating-point computation that today are *bandwidth-limited*, by the ability of the CPU to traverse large matrices in RAM.

The authors' solution is to develop a compressed representation for matrices, as well as compressed linear algebra operations that work directly over the compressed matrix data. Together, these reduce the bandwidth required to perform the same computations, thus speeding performance dramatically. The paper cleverly adapts ideas first developed in relational database systems — column-oriented compression, sampling-based cost estimation, trading between compression speed and compression rate — to build an elegant implementation.

The paper makes a number of key contributions. First, the authors identify a set of linear algebra primitives shared by multiple distributed machine learning platforms and algorithms. Second, they develop compression techniques not only for single columns in a matrix, but also "column grouping" techniques that capitalize on correlations between columns. They show that offset lists and run-length encoding offer a good set of trade-offs between efficiency and performance. Third, the paper develops cache-conscious algorithms for matrix multiplication and other operations. Finally, the paper shows how to estimate the sizes of compressed matrices and to choose an effective compression strategy. Together, these techniques illustrate how database systems concepts can be adapted to great benefit in the machine learning space.

References

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. A. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. Tensorflow: A system for large-scale machine learning. In *OSDI*, pages 265–283, 2016.

[2] A. Elgohary, M. Boehm, P. J. Haas, F. R. Reiss, and B. Reinwald. Compressed linear algebra for large-scale machine learning. *PVLDB*, 9(12):960–971, 2016.

[3] S. Ewen, K. Tzoumas, M. Kaufmann, and V. Markl. Spinning fast iterative data flows. *Proc. VLDB Endow.*, 5(11):1268–1279, 2012.

[4] A. Ghoting, R. Krishnamurthy, E. P. D. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian, and S. Vaithyanathan. Systemml: Declarative machine learning on mapreduce. In *ICDE*, pages 231–242, 2011.

[5] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *PVLDB*, 2012.

[6] Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, May 2016.

[7] M. Zaharia, M. Chodhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *HotCloud*, 2010.

Scaling Machine Learning via Compressed Linear Algebra

Ahmed Elgohary[2] Matthias Boehm[1], Peter J. Haas[1], Frederick R. Reiss[1], Berthold Reinwald[1]

[1] IBM Research – Almaden; San Jose, CA, USA
[2] University of Maryland; College Park, MD, USA

ABSTRACT

Large-scale machine learning (ML) algorithms are often iterative, using repeated read-only data access and I/O-bound matrix-vector multiplications to converge to an optimal model. It is crucial for performance to fit the data into single-node or distributed main memory and enable very fast matrix-vector operations on in-memory data. General-purpose, heavy- and lightweight compression techniques struggle to achieve both good compression ratios and fast decompression speed to enable block-wise uncompressed operations. Compressed linear algebra (CLA) avoids these problems by applying lightweight lossless database compression techniques to matrices and then executing linear algebra operations such as matrix-vector multiplication directly on the compressed representations. The key ingredients are effective column compression schemes, cache-conscious operations, and an efficient sampling-based compression algorithm. Experiments on an initial implementation in SystemML show in-memory operations performance close to the uncompressed case and good compression ratios. We thereby obtain significant end-to-end performance improvements up to 26x or reduced memory requirements.

1. INTRODUCTION

Large-scale machine learning (ML) leverages large data collections in order to find interesting patterns and build robust predictive models [9, 10]. Applications include regression analysis, classification, and recommendations. Data-parallel frameworks such as MapReduce [11], Spark [22], and Flink [2] are often used for cost-effective parallelized model building on commodity hardware.

Declarative ML: State-of-the-art, large-scale ML systems support declarative ML algorithms [5], expressed in high-level languages, that comprise linear algebra operations, i.e., matrix multiplications, aggregations, element-wise and statistical computations. Examples—at varying abstraction levels—are SystemML [6], SciDB [20], Cumulon [15], DMac [21], and TensorFlow [1]. A high level of abstraction gives data scientists the flexibility to create and customize ML algorithms without worrying about data and cluster characteristics, underlying data representations (e.g., sparse/dense format) or execution plan generation. We aim to improve the performance of declarative ML algorithms.

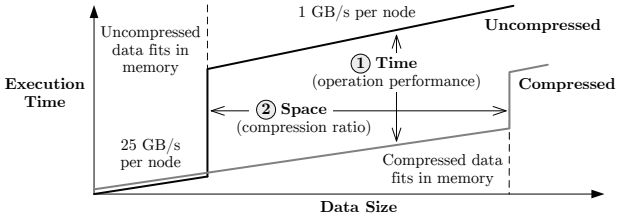


Figure 1: Goals of Compressed Linear Algebra.

Bandwidth Challenge: Many ML algorithms are iterative, with repeated read-only access to the data. They rely on matrix-vector multiplications to converge to an optimal model; such operations require one complete scan of the matrix, with two floating point operations per matrix element. Hence, matrix-vector multiplications are, even in-memory, I/O bound. Disk bandwidth is usually 10x-100x slower than memory bandwidth, so it it crucial for performance to fit the matrix into available memory without sacrificing operations performance. This challenge applies to single-node in-memory computations [16], data-parallel frameworks with distributed caching such as Spark [22], and hardware accelerators like GPUs, with limited device memory [1, 3, 4].

Compressed Linear Algebra: Declarative ML provides data independence, which allows for automatic lossless compression to fit larger datasets into memory. A baseline solution would employ general-purpose compression techniques and decompress matrices block-wise for each operation. However, heavyweight techniques like Gzip are inapplicable because decompression is too slow (slower than uncompressed I/O), while lightweight methods like Snappy only achieve moderate compression ratios. Existing special-purpose compressed matrix formats with good performance like CSR-VI [18] similarly show only modest compression ratios. We have therefore initiated the study of *compressed linear algebra (CLA)*, in which lightweight database compression methods—such as compressing offset lists per distinct column value—are applied to matrices and then linear algebra operations are executed directly on the compressed representations [12]. Figure 1 shows the goals of this approach: we want to widen the sweet spot for compression by achieving *both* (1) performance close to uncompressed in-memory operations and (2) good compression ratios to fit larger datasets into memory. The novelty of our approach is to combine both database compression techniques and sparse matrix representations, leading towards a generalization of traditional sparse matrix formats for sparse and dense data; see [12] for a full discussion of related work.

Table 1: Compression Ratios of Real Datasets.

Dataset	Size	Gzip	Snappy	CLA
Higgs [19]	11M×28, 0.92: 2.5 GB	1.93	1.38	**2.03**
Census [19]	2.5M×68, 0.43: 1.3 GB	17.11	6.04	**27.46**
Covtype [19]	600K×54, 0.22: .14 GB	10.40	6.13	**12.73**
ImageNet [8]	1.2M×900, 0.31: 4.4 GB	5.54	3.35	**7.38**
Mnist8m [7]	8.1M×784, 0.25: 19 GB	4.12	2.60	**6.14**

Table 2: Overview ML Algorithm Core Operations (see `http://systemml.apache.org/algorithms` for details).

Algorithm	M-V $\mathbf{Xv}$	V-M $\mathbf{v}^\top\mathbf{X}$	MVChain $\mathbf{X}^\top(\mathbf{w}\odot(\mathbf{Xv}))$	TSMM $\mathbf{X}^\top\mathbf{X}$
LinregCG	✓	✓	✓ (w/o $\mathbf{w}\odot$)	
LinregDS		✓		✓
Logreg / GLM	✓	✓	✓ (w/ $\mathbf{w}\odot$)	
L2SVM	✓	✓		
PCA	✓			✓

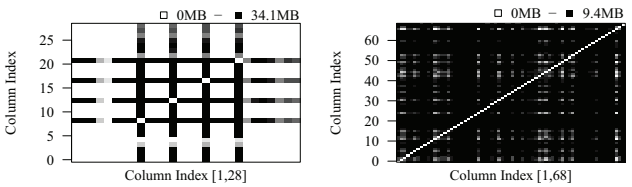
Compression Potential: Our focus is on floating-point matrices, so the potential for compression may not be obvious. Table 1 shows compression ratios for the general-purpose, heavyweight Gzip and lightweight Snappy algorithms and for our CLA method on real-world datasets (sizes given as rows, columns, sparsity, and in-memory size). We see compression ratios of 2x-27x, due to the presence of a mix of floating point and integer data, and due to features with relatively few distinct values. Thus enterprise machine-learning datasets are indeed amenable to compression. The decompression bandwidth (including time for matrix deserialization) of Gzip ranges from 88 MB/s to 291 MB/s which is slower than for uncompressed I/O. Snappy achieves a decompression bandwidth between 232 MB/s and 638 MB/s but only moderate compression ratios. In contrast, CLA achieves good compression ratios and avoids decompression. In the following sections, we motivate our approach and describe its key components: column compression schemes, cache-conscious vector-matrix operations, and an efficient sampling-based compression algorithm.

2. BACKGROUND AND MOTIVATION

As discussed below, both the features of declarative-ML systems and the characteristics of typical ML workloads motivate our approach to compressed linear algebra.

SystemML Setting: We describe CLA in the setting of SystemML, as it is representative of the declarative ML platforms that we are targeting. In SystemML, algorithms are expressed in a high-level R-like scripting language and compiled to hybrid runtime plans that combine both single-node, in-memory operations and distributed operations on MapReduce or Spark. Each statement block of an ML script is first parsed into a directed cyclic graph (DAG) of high-level operators. The system then applies various rewrites, such as common subexpression elimination and optimization of matrix-multiplication chains, as well as operator selection, yielding a DAG of low-level operators, which is then compiled into instructions. Matrices are represented internally in a binary *block matrix* format with fixed-size blocks. Each block is represented either in dense or sparse format. For single-node, in-memory operations, the entire matrix is often represented as a single block. CLA can be seamlessly integrated by adding a new derived block representation and operations. See [6, 12] for further details on SystemML.

Common Operation Characteristics: Table 2 summarizes the core operations of important ML algorithms.



(a) Co-Coding Higgs (b) Co-Coding Census

Figure 2: Cardinality Ratios and Co-Coding.

These algorithms include linear regression via iterative conjugate-gradient descent (LinregCG) and via direct solution of the normal equations (LinregDS), as well as logistic regression (Logreg), generalized linear models (GLM), support-vector machines with L_2-regularization (L2SVM) and principal component analysis (PCA). LinregDS and PCA are non-iterative and the other algorithms are iterative. Vector-matrix multiplication is often caused by the rewrite $\mathbf{X}^\top\mathbf{v} \rightarrow (\mathbf{v}^\top\mathbf{X})^\top$ to avoid transposing $\mathbf{X}$. In addition, many systems also implement physical operators for matrix-vector chains, with optional element-wise weighting $\mathbf{w}\odot$, and transpose-self matrix multiplication (TSMM) $\mathbf{X}^\top\mathbf{X}$. All of these operations are I/O-bound, except for TSMM with $m \gg 1$ features because its compute workload grows as $O(m^2)$. Beside these operations, `append`, unary aggregates like `colSums`, and matrix-scalar operations access $\mathbf{X}$ for intercept computation, scaling and shifting.

Common Data Characteristics: Despite significant differences in data sizes—ranging from kilobytes to terabytes—we and others have observed certain common characteristics of ML datasets. First, matrices usually have significantly more rows (observations) than columns (features), especially in enterprise machine learning, where data often originates from data warehouses. Second, feature preprocessing like dummy coding often yields datasets having many *sparse* features (i.e., features with many zero values); sparsity, however, is rarely uniform, but often varies widely among features [12]. Third, Many datasets contain features with low *column cardinality*, i.e., few distinct values. Examples include encoded categorical, binned or dummy-coded (0/1) features. Low column cardinality is a good indicator of compression potential because it indicates redundancy. For example, all columns of *Census* have a ratio of column cardinality to number of rows below .0008% and the majority of columns of *Higgs* have a cardinality ratio below 1%. The column cardinalities can vary widely within a dataset, however; for example, *Higgs* contains several columns having millions of distinct values. (See [12] for additional discussion of the datasets.) Finally, many datasets contain column groups that exhibit significant *correlation* in that the concatenated columns have a cardinality ratio much lower than would be expected if the values in each column were arranged randomly and independently of the other columns.

Compression Strategy: The foregoing workload characteristics suggest several key features of a good compression strategy. First, the compression schemes should be column-based and value-centric, with fallbacks for high cardinality columns. Moreover, schemes should exploit column correlation by discovering and *co-coding* highly correlated column groups. With value-based offset lists, a column i with d_i distinct values requires $\approx 8d_i + 4n$ B, where n is the number of rows, and each value is encoded with 8 B and a list of 4 B row indexes. Co-coding two columns i and j as a single

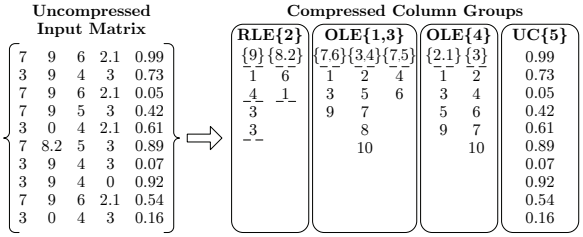


Figure 3: Example Compressed Matrix Block.

group of value-pairs and offsets requires $16d_{ij} + 4n$ B, where d_{ij} is the number of distinct value-pairs. The higher the correlation, the larger the size reduction by co-coding. For example, Figures 2(a) and 2(b) show the size reductions (in MB) by co-coding all pairs of columns of *Higgs* and *Census*. Overall, co-coding column groups of *Census* (not limited to pairs) improves the compression ratio from 10.1x to 27.4x. For *Higgs*, co-coding any of the columns 8, 12, 16, and 20 with one of *most* of the other columns reduces sizes by at least 25 MB. Moreover, co-coding *any* column pair of *Census* reduces sizes by at least 9.3 MB.

3. COMPRESSION SCHEMES

We now describe our novel matrix compression framework, including two effective encoding formats for compressed column groups, as well as efficient, cache-conscious core linear algebra operations over compressed matrices.

3.1 Matrix Compression Framework

A compressed matrix block is represented as a set of compressed columns. Column-wise compression leverages two key characteristics: few distinct values per column and high cross-column correlations. Taking advantage of few distinct values, we encode a column as a list of distinct values together with a list of *offsets* per value, i.e., a list of row indexes in which the value appears. As with sparse matrix formats, offset lists allow for efficient linear algebra operations.

Column Co-Coding: We exploit column correlation by partitioning columns into *column groups* such that columns within each group are highly correlated. Columns within the same group are then co-coded as a single unit. Conceptually, each row of a column group comprising m columns is an m-tuple $\mathbf{t}$ of floating-point values, representing reals or integers.

Column Encoding Formats: Conceptually, the offset list associated with each distinct tuple is stored as a compressed sequence of bytes. The efficiency of executing linear algebra operations over compressed matrices strongly depends on how fast we can iterate over this compressed representation. We adapt two well-known effective offset-list encoding formats: Offset-List Encoding (OLE) and Run-Length Encoding (RLE). We fall back to a simple uncompressed-column (UC) format if compression is not beneficial. These decisions on column encoding formats as well as co-coding are strongly data-dependent and hence require automatic optimization. We discuss compression planning—i.e., automatically choosing plans that maximize the compression ratio—in Section 4.

Example Compressed Matrix: Figure 3 shows our running example of a compressed matrix block. The 10×5 input matrix is represented as four column groups. Columns 2, 4, and 5 are represented as single-column groups and encoded with RLE, OLE, and UC, respectively. For column 4, we have two distinct non-zero values and hence two offset lists. Finally, there is a co-coded column group for the correlated columns 1 and 3, which encodes offset lists for all distinct value-pairs.

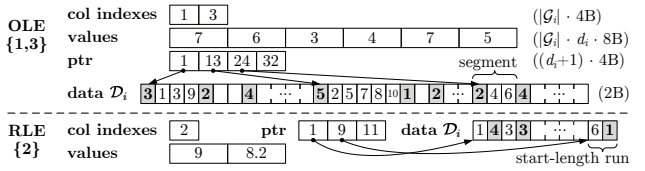


Figure 4: Data Layout OLE/RLE Column Groups.

Notation: For the ith column group, denote by $\mathcal{T}_i = \{\mathbf{t}_{i1}, \mathbf{t}_{i2}, \ldots, \mathbf{t}_{id_i}\}$ the set of d_i distinct non-zero tuples, by $\mathcal{G}_i$ the set of column indexes, and by $\mathcal{O}_{ij}$ the set of offsets associated with $\mathbf{t}_{ij}$ $(1 \le j \le d_i)$. We focus on the "sparse" case in which zero values are not stored (aka "0-suppressing"). Also, denote by α the size in bytes of each floating point value; $\alpha = 8$ for the double-precision IEEE-754 standard.

3.2 Column Encoding Formats

We now describe the compressed data layout of the OLE and RLE formats and give formulas for the in-memory compressed size S_i^{OLE} and S_i^{RLE}. The total matrix size is then computed as the sum of group size estimates.

Data Layout: Figure 4 shows—as an extension to our running example from Figure 3 (with more rows)—the data layout of OLE/RLE column groups composed of four linearized arrays. Besides a data array $\mathcal{D}_i$, both encoding schemes use a common header of three arrays for column indexes, fixed-length value tuples, and pointers to the data per tuple. The physical data length per tuple in $\mathcal{D}_i$ can be computed as the difference of adjacent pointers (e.g., for $\mathbf{t}_{i1} = \{7, 6\}$ as $13 - 1 = 12$). The data array is then used in an encoding-specific manner. Tuples are stored in order of decreasing physical data length to improve branch prediction and pre-fetching.

Offset-List Encoding (OLE): Our OLE scheme divides the offset range into *segments* of fixed length $\Delta^{\text{s}} = 2^{16}$ (two bytes per offset). Each offset is mapped to its corresponding segment and encoded as the difference to the beginning of its segment. For example, the offset 155,762 lies in segment 3 $(= 1 + \lfloor (155{,}762 - 1)/\Delta^{\text{s}} \rfloor)$ and is encoded as 24,690 $(= 155{,}762 - 2\Delta^{\text{s}})$. Each segment then encodes the number of offsets with two bytes, followed by two bytes for each offset, resulting in a variable physical length in $\mathcal{D}_i$. Empty segments are represented as two bytes indicating zero length. Iterating over an OLE group entails scanning the segmented offset list and reconstructing global offsets as needed. The size S_i^{OLE} of column group $\mathcal{G}_i$ is calculated as

$$S_i^{\text{OLE}} = 4|\mathcal{G}_i| + d_i\big(4 + \alpha|\mathcal{G}_i|\big) + 2\sum_{j=1}^{d_i} b_{ij} + 2z_i, \qquad (1)$$

where b_{ij} denotes the number of segments of tuple $\mathbf{t}_{ij}$, $|\mathcal{O}_{ij}|$ denotes the number of offsets for $\mathbf{t}_{ij}$, and $z_i = \sum_{j=1}^{d_i} |\mathcal{O}_{ij}|$ denotes the total number of offsets in the column group. The common header has a size of $4|\mathcal{G}_i| + d_i\big(4 + \alpha|\mathcal{G}_i|\big)$.

Run-Length Encoding (RLE): In RLE, a sorted list of offsets is encoded as a sequence of *runs*. Each run represents a consecutive sequence of offsets, via two bytes for the starting offset and two bytes for the run length. We store starting offsets as the difference between the offset and the ending offset of the preceding run. Empty runs are used when a

Algorithm 1 Cache-Conscious OLE Matrix-Vector

Input: OLE column group $\mathcal{G}_i$, vectors $\mathbf{v}$, $\mathbf{q}$, row range $[rl, ru)$
Output: Modified vector $\mathbf{q}$ (in row range $[rl, ru)$)

```
1: for j in [1, d_i] do                                  // distinct tuples
2:    π_ij ← SKIPSCAN(G_i, j, rl)                        // find position of rl in D_i
3:    u_ij ← t_ij · v_{G_i}                              // pre-aggregate value
4: for bk in [rl, ru) by Δ^c do                          // cache buckets in [rl, ru)
5:    for j in [1, d_i] do                               // distinct tuples
6:       for k in [bk, min(bk + Δ^c, ru)) by Δ^s do      // segments
7:          if π_ij ≤ b_ij + |O_ij| then                // physical data length
8:             ADDSEGMENT(G_i, π_ij, u_ij, k, q)        // update q, π_ij
```

relative starting offset is larger than the maximum length of 2^{16}. Similarly, runs exceeding the maximum length are partitioned into smaller runs. Iterating over an RLE group entails scanning the runs and enumerating offsets per run. The size S_i^{RLE} of column group $\mathcal{G}_i$ is computed as

$$S_i^{\text{RLE}} = 4|\mathcal{G}_i| + d_i\big(4 + \alpha|\mathcal{G}_i|\big) + 4\sum_{j=1}^{d_i} r_{ij}, \qquad (2)$$

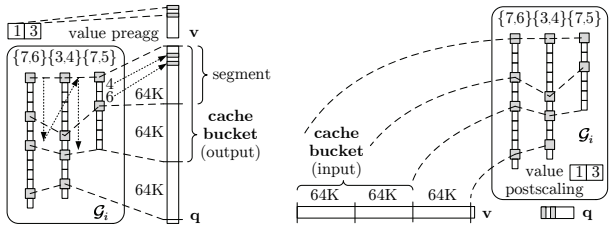
where r_{ij} is the number of runs for tuple $\mathbf{t}_{ij}$. Again, the common header has a size of $4|\mathcal{G}_i| + d_i\big(4 + \alpha|\mathcal{G}_i|\big)$.

3.3 Operations over Compressed Matrices

We now show how to execute efficient linear algebra operations over a set $\mathcal{X}$ of column groups; matrix block operations are then composed of operations over column groups. We write $c\mathbf{v}$ to denote element-wise scalar-vector multiplication as well as $\mathbf{u} \cdot \mathbf{v}$ to denote the inner product of vectors.

Matrix-Vector Multiplication: The product $\mathbf{q} = \mathbf{X}\mathbf{v}$ of $\mathbf{X}$ and a column vector $\mathbf{v}$ can be represented with respect to column groups as $\mathbf{q} = \sum_{i=1}^{|\mathcal{X}|}\sum_{j=1}^{d_i}(\mathbf{t}_{ij} \cdot \mathbf{v}_{\mathcal{G}_i})\mathbf{1}_{\mathcal{O}_{ij}}$, where $\mathbf{v}_{\mathcal{G}_i}$ is the projection of $\mathbf{v}$ onto the indexes $\mathcal{G}_i$ and $\mathbf{1}_{\mathcal{O}_{ij}}$ is the 0/1-indicator vector of offset list $\mathcal{O}_{ij}$. A straightforward way to implement this computation iterates over $\mathbf{t}_{ij}$ tuples in each group, scanning $\mathcal{O}_{ij}$ and adding $\mathbf{t}_{ij} \cdot \mathbf{v}_{\mathcal{G}_i}$ at reconstructed offsets to $\mathbf{q}$. However, pure column-wise processing would scan the $n \times 1$ output vector $\mathbf{q}$ once per tuple, resulting in cache-unfriendly behavior for the typical case of large n. We therefore use cache-conscious schemes for OLE and RLE groups based on *horizontal, segment-aligned scans* (with benefits of up to 2.1x/5.4x for M-V/V-M in our experiments); see Algorithm 1 and Figure 5(a) for the case of OLE. Multi-threaded operations parallelize over segment-aligned partitions of rows $[rl, ru)$, which guarantees disjoint results and thus avoids partial results per thread. We find π_{ij}, the starting position of each $\mathbf{t}_{ij}$ in $\mathcal{D}_i$ via a skip scan that aggregates segment lengths until we reach rl (line 2). To minimize the overhead of finding π_{ij}, we use static scheduling (task partitioning). We further pre-compute $\mathbf{u}_{ij} = \mathbf{t}_{ij} \cdot \mathbf{v}_{\mathcal{G}_i}$ once for all tuples (line 3). For each cache-bucket of size Δ^c (such that $\Delta^c \cdot \#\text{cores} \cdot 8\,\text{B}$ fits in L3 cache, by default $\Delta^c = 2\Delta^s$), we then iterate over all distinct tuples (lines 5-8) but maintain the current positions π_{ij} as well. The inner loop (lines 6-8) then scans segments and adds $\mathbf{u}_{ij}$ via scattered writes at reconstructed offsets to the output $\mathbf{q}$ (line 8). RLE is similarly realized except for sequential writes to $\mathbf{q}$ per run, special handling of partition boundaries, and additional state for the reconstructed start offsets per tuple.

As a toy example for OLE, consider the column group $\mathcal{G} = \{1, 3\}$ as in Figure 4 and suppose that $\mathbf{v}_{\mathcal{G}} = (1, 2)$. Also suppose that the OLE encoding uses two segments, each of length = 5 rows, and that a cache bucket comprises exactly one segment. Finally, suppose that a single thread



(a) Matrix-Vector (b) Vector-Matrix

Figure 5: Cache-Conscious OLE Operations.

updates $\mathbf{q}$. Algorithm 1 first precomputes $(1, 2) \cdot (7, 6) = 19$, $(1, 2) \cdot (3, 4) = 11$, and $(1, 2) \cdot (7, 5) = 17$. The thread then handles rows in $[rl, ru) = [1, 11)$, i.e., all ten rows. It reads the first five elements of $\mathbf{q}$ into cache, and then adds 19 to $\mathbf{q}[1]$ and $\mathbf{q}[3]$, 11 to $\mathbf{q}[2]$ and $\mathbf{q}[5]$, and 17 to $\mathbf{q}[4]$. Next, the thread reads in the last five elements of $\mathbf{q}$ and adds 19 to $\mathbf{q}[9]$, 11 to $\mathbf{q}[7]$, $\mathbf{q}[8]$, $\mathbf{q}[10]$, and 17 to $\mathbf{q}[6]$. In contrast, the naïve approach would first add 19 to $\mathbf{q}[i]$ for $i = 1, 3, 9$, then add 11 to $\mathbf{q}[i]$ for $i = 2, 5, 7, 8, 10$, and then add 17 to $\mathbf{q}[i]$ for $i = 4, 6$. The cost on our toy architecture is six "cache reads" compared to two reads for Algorithm 1. Also note that Algorithm 1 requires only 6 multiplications and 13 additions, whereas the uncompressed operation requires 20 multiplications and 20 additions.

Vector-Matrix Multiplication: Column-wise compression allows for efficient vector-matrix products $\mathbf{q} = \mathbf{v}^\top\mathbf{X}$ because individual column groups update disjoint entries of the output vector $\mathbf{q}$. Each entry q_i can be expressed over columns as $q_i = \mathbf{v}^\top\mathbf{X}_{:i}$. We rewrite this multiplication in terms of a column group $\mathcal{G}_i$ as scalar-vector multiplications: $\mathbf{q}_{\mathcal{G}_i} = \sum_{j=1}^{d_i}\sum_{l \in \mathcal{O}_{ij}} v_l\mathbf{t}_{ij}$. However, a purely column-wise processing would again suffer from cache-unfriendly behavior because we scan the input vector $\mathbf{v}$ once for each distinct tuple. Our cache-conscious OLE/RLE group operations again use *horizontal, segment-aligned scans* as shown in Figure 5(b). The OLE/RLE algorithms are similar to matrix-vector but in the inner loop we sum up input-vector values according to the given offset list; finally, we scale the aggregated value once with the values in $\mathbf{t}_{ij}$. For multi-threaded operations, we parallelize over column groups, where disjoint results per column allow for simple dynamic task scheduling. The cache bucket size is equivalent to matrix-vector (by default $2\Delta^s$) except that RLE runs are allowed to cross cache bucket boundaries due to column-wise parallelization.

Other Operations: As discussed in [12], efficient methods for more complex operations such as matrix-vector multiplication chains and transpose-self matrix multiplications are built up from the foregoing matrix-vector and vector-matrix operations. Common operations such as $\mathbf{X}^2$, $2\mathbf{X}$, and `append` can be executed very efficiently over compressed matrices without scanning the offset lists. Finally, unary aggregates like `sum` (or similarly `colSums`) are efficiently computed using offset-list sizes as $\sum_{i=1}^{|\mathcal{X}|}\sum_{j=1}^{d_i}|\mathcal{O}_{ij}|\mathbf{t}_{ij}$.

4. COMPRESSION PLANNING

Given an uncompressed $n \times m$ matrix block $\mathbf{X}$, the system automatically chooses a compression plan, that is, a partitioning of compressible columns into column groups and a compression scheme per group. To keep the planning costs low, sampling-based techniques are used to estimate

the compressed size of an OLE/RLE column group $\mathcal{G}_i$. The size estimates are used for finding the initial set of compressible columns and a good column-group partitioning. Exhaustive ($O(m^m)$) and brute-force greedy ($O(m^3)$) partitioning are infeasible, but a bin-packing-based technique can drastically reduce the number of candidate groups. The overall compression algorithm corrects for estimation errors.

4.1 Estimating Compressed Size

To calculate the compressed size of a column group $\mathcal{G}_i$ via size-estimation formulas (1) and (2), we need to estimate the number of distinct tuples d_i, non-zero tuples z_i, segments b_{ij}, and runs r_{ij}. Our estimators are based on a small sample of rows $\mathcal{S}$ drawn randomly and uniformly from $\mathbf{X}$ with $|\mathcal{S}| \ll n$. We have found experimentally that being conservative (overestimating compressed size) and correcting later on yields the most robust co-coding choices, so we make conservative choices in our estimator design.

Number of Distinct Tuples: To estimate d_i, we use the "hybrid" estimator $\hat{d}_i$ from [14]; the idea is to estimate the degree of variability in the frequencies of the tuples in $\mathcal{T}_i$ as low, medium, or high, based on the estimated squared coefficient of variation and then apply a "generalized jackknife" estimator that performs well for that regime. Such an estimator has the general form $\hat{d} = d_{\mathcal{S}} + K(N^{(1)}/|\mathcal{S}|)$, where $d_{\mathcal{S}}$ is the number of distinct tuples in the sample, K is a data-based constant, and $N^{(1)}$ is the number of tuples that appear exactly once in $\mathcal{S}$ ("singletons"). The hybrid estimator provides a reasonable balance of cost and accuracy [12].

Number of OLE Segments: In general, not all elements of $\mathcal{T}_i$ will appear in the sample. Denote by $\mathcal{T}_i^o$ and $\mathcal{T}_i^u$ the sets of tuples observed and unobserved in the sample, and by d_i^o and d_i^u their cardinalities. The latter can be estimated as $\hat{d}_i^u = \hat{d}_i - d_i^o$, where $\hat{d}_i$ is obtained as described above. We also need to estimate the population frequencies of both observed and unobserved tuples. Let f_{ij} be the population frequency of tuple $\mathbf{t}_{ij}$ and F_{ij} the sample frequency. A naïve estimate scales up F_{ij} to obtain $f_{ij}^{\text{naïve}} = (n/|\mathcal{S}|)F_{ij}$. Note that $\sum_{\mathbf{t}_{ij} \in \mathcal{T}_i^o} f_{ij}^{\text{naïve}} = n$ implies a zero population frequency for each unobserved tuple. We adopt a standard way of dealing with this issue and scale down the naïve frequency estimates by the estimated "coverage" C_i of the sample, defined as $C_i = \sum_{\mathbf{t}_{ij} \in \mathcal{T}_i^o} f_{ij}/n$. The usual estimator of coverage, originally due to Turing (see [13]), is

$$\hat{C}_i = \max\big(1 - N_i^{(1)}/|\mathcal{S}|, |\mathcal{S}|/n\big). \tag{3}$$

This estimator assumes a frequency of one for unseen tuples, computing the coverage as one minus the fraction of singletons in the sample. We add the lower sanity bound $|\mathcal{S}|/n$ to handle the case $N_i^{(1)} = |\mathcal{S}|$. For simplicity, we assume equal frequencies for all unobserved tuples. The resulting frequency estimation formula for tuple $\mathbf{t}_{ij}$ is

$$\hat{f}_{ij} = \begin{cases} (n/|\mathcal{S}|)\hat{C}_i F_{ij} & \text{if } \mathbf{t}_{ij} \in \mathcal{T}_i^o \\ n(1-\hat{C}_i)/\hat{d}_i^u & \text{if } \mathbf{t}_{ij} \in \mathcal{T}_i^u. \end{cases} \tag{4}$$

We can now estimate the number of segments b_{ij} in which tuple $\mathbf{t}_{ij}$ appears at least once (this modified definition of b_{ij} ignores empty segments for simplicity with negligible error in our experiments). There are $l = n - |\mathcal{S}|$ unobserved offsets and estimated $\hat{f}_{iq}^u = \hat{f}_{iq} - F_{iq}$ unobserved instances of tuple $\mathbf{t}_{iq}$ for each $\mathbf{t}_{iq} \in \mathcal{T}_i$. We adopt a maximum-entropy (maxEnt) approach and assume that all assignments of unobserved tuple instances to unobserved offsets are equally likely. Denote by $\mathcal{B}$ the set of segment indexes and by $\mathcal{B}_{ij}$ the subset of indexes corresponding to segments with at least one observation of $\mathbf{t}_{ij}$. Also, for $k \in \mathcal{B}$, let l_k be the number of unobserved offsets in the kth segment and N_{ijk} the random number of unobserved instances of $\mathbf{t}_{ij}$ assigned to the kth segment ($N_{ijk} \leq l_k$). Then we estimate b_{ij} by its expected value under our maxEnt model:

$$\begin{aligned} \hat{b}_{ij} = E[b_{ij}] &= |\mathcal{B}_{ij}| + \sum_{k \in \mathcal{B} \setminus \mathcal{B}_{ij}} P(N_{ijk} > 0) \\ &= |\mathcal{B}_{ij}| + \sum_{k \in \mathcal{B} \setminus \mathcal{B}_{ij}} [1 - h(l_k, \hat{f}_{ij}^u, l)], \end{aligned} \tag{5}$$

where $h(a,b,c) = \binom{c-b}{a} / \binom{c}{a}$ is a hypergeometric probability. Note that $\hat{b}_{ij} \equiv \hat{b}_i^u$ for $\mathbf{t}_{ij} \in \mathcal{T}_i^u$, where $\hat{b}_i^u$ is the value of $\hat{b}_{ij}$ when $\hat{f}_{ij}^u = (1-\hat{C}_i)n/\hat{d}_i^u$ and $|\mathcal{B}_{ij}| = 0$. Thus our estimate of the sum $\sum_{j=1}^{d_i} b_{ij}$ in (1) is $\sum_{\mathbf{t}_{ij} \in \mathcal{T}_i^o} \hat{b}_{ij} + \hat{d}_i^u \hat{b}_i^u$.

Number of Non-Zero Tuples: We estimate the number of non-zero tuples as $\hat{z}_i = n - \hat{f}_{i0}$, where $\hat{f}_{i0}$ is an estimate of the number of zero tuples in $\mathbf{X}_{:\mathcal{G}_i}$. Denote by F_{i0} the number of zero tuples in the sample. If $F_{i0} > 0$, we can proceed as above and set $\hat{f}_{i0} = (n/|\mathcal{S}|)\hat{C}_i F_{i0}$, where $\hat{C}_i$ is (3). If $F_{i0} = 0$, then we set $\hat{f}_{i0} = 0$; this estimate maximizes $\hat{z}_i$ and hence $\hat{S}_i^{\text{OLE}}$ per our conservative estimation strategy.

Number of RLE Runs: The number of RLE runs r_{ij} for tuple $\mathbf{t}_{ij}$ is estimated as the expected value of r_{ij} under the maxEnt model. This expected value is very hard to compute exactly and Monte Carlo approaches are too expensive, so we approximate $E[r_{ij}]$ by considering one interval of consecutive unobserved offsets at a time as shown in Figure 6. Adjacent intervals are separated by a "border" comprising one or more observed offsets. As with the OLE estimates, we ignore the effects of empty and very long runs. Denote by η_k the length of the kth interval and set $\eta = \sum_k \eta_k$. Under the maxEnt model, the number f_{ijk}^u of unobserved $\mathbf{t}_{ij}$ instances assigned to the kth interval is hypergeometric, and we estimate f_{ijk}^u by its mean value: $\hat{f}_{ijk}^u = (\eta_k/\eta)\hat{f}_{ij}^u$. Given that $\hat{f}_{ijk}^u$ instances of $\mathbf{t}_{ij}$ are assigned randomly and uniformly among the η_k possible positions in the interval, the number of runs r_{ijk} within the interval (ignoring the borders) is known to follow an "Ising-Stevens" distribution [17, pp. 422-423] and we estimate r_{ijk} by its mean: $\hat{r}_{ijk} = \hat{f}_{ijk}^u(\eta_k - \hat{f}_{ijk}^u + 1)/\eta_k$. We show in [12] that a reasonable estimate of the contribution to r_{ij} from the border between interior intervals k and $k+1$ is $\hat{A}_{ijk} = 1 - (2\hat{f}_{ij}^u/\eta)$, so that the final estimate is $\hat{r}_{ij} = \sum_k \hat{r}_{ijk} + \sum_k \hat{A}_{ijk}$ (with appropriate modifications for the first and last border).

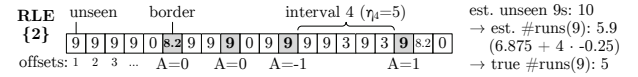


Figure 6: Estimating the Number of RLE Runs $\hat{r}_{ij}$.

4.2 Partitioning Columns into Groups

A greedy brute-force method for partitioning a set of compressible columns into groups starts with singleton groups and executes merging iterations. At each iteration, we merge the two groups having maximum compression ratio (sum of their compressed sizes divided by the compressed size of the merged group). We terminate when no further space reductions are possible, i.e., no compression ratio exceeds 1. Al-

Algorithm 2 Matrix Block Compression

Input: Matrix block $\mathbf{X}$ of size $n \times m$
Output: A set of compressed column groups $\mathcal{X}$

```
1: C^C ← ∅, C^UC ← ∅, G ← ∅, X ← ∅
2: // Planning phase - - - - - - - - - - - - - - - - - - -
3: S ← sampleRowsUniform(X, sample_size)
4: for all column k in X do                          // classify
5:    cmp_ratio ← ẑ_i α / min(Ŝ_k^RLE, Ŝ_k^OLE)
6:    if cmp_ratio > 1 then
7:       C^C ← C^C ∪ k
8:    else
9:       C^UC ← C^UC ∪ k
10: bins ← runBinPacking(C^C)                        // group
11: for all bin b in bins do
12:    G ← G ∪ groupBruteForce(b)
13: // Compression phase - - - - - - - - - - - - - - - - - -
14: for all column group G_i in G do                 // compress
15:    do
16:       biglist ← extractBigList(X, G_i)
17:       cmp_ratio ← getExactCmpRatio(biglist)
18:       if cmp_ratio > 1 then
19:          X ← X ∪ compressBigList(biglist), break
20:       k ← removeLargestColumn(G_i)
21:       C^UC ← C^UC ∪ k
22:    while |G_i| > 0
23: return X ← X ∪ createUCGroup(C^UC)
```

though compression ratios are estimated from a sample, the cost of the brute-force scheme is $O(m^3)$, which is infeasible.

Bin Packing: We observed empirically that the brute-force method usually generates groups of no more than five columns. Further, we noticed that the time needed to estimate a group size increases as the sample size, the number of distinct tuples, or the matrix density increases. These two observations motivate a heuristic strategy where we partition the columns into a set of small *bins* and then apply the brute-force method within each bin to form the column groups. We use a bin-packing algorithm to assign columns to bins. The weight of each column indicates its estimated contribution to the overall runtime of the brute-force partitioning. The capacity of a bin is chosen to ensure moderate brute-force runtime per bin. Intuitively, bin packing minimizes the number of bins, which should maximize the number of columns within each bin and hence grouping potential, while controlling the processing costs.

Bin Weights: We set the weight of the i^{th} column to $\hat{d}_i/n$, i.e., the ratio of distinct tuples to rows. If $\hat{d}_i/n$ is larger than a specified threshold γ, then we consider column i as ineligible for grouping. We also set each bin capacity to $w = \beta\gamma$, where β is a tuning parameter. We made the design choice of a constant bin capacity—independent of the number of non-zeros—to ensure constant compression ratios and throughput irrespective of blocking configurations. We use the first-fit heuristic to solve the bin-packing problem.

4.3 Compression Algorithm

We now describe the overall algorithm for creating compressed matrix blocks (Algorithm 2). Note that we transpose the input in case of row-major dense or sparse formats to avoid performance issues due to column-wise processing.

Planning Phase (lines 2-12)**:** Planning starts by drawing a sample of rows from $\mathbf{X}$. For each column i, the sample is first used to estimate the compressed column size S_i^{C} by $\hat{S}_i^{\text{C}} = \min(\hat{S}_i^{\text{RLE}}, \hat{S}_i^{\text{OLE}})$, where $\hat{S}_i^{\text{RLE}}$ and $\hat{S}_i^{\text{OLE}}$ are obtained by substituting the estimated $\hat{d}_i$, $\hat{z}_i$, $\hat{r}_{ij}$, and $\hat{b}_{ij}$ into formulas (1) and (2). We conservatively estimate the uncompressed column size as $\hat{S}_i^{\text{UC}} = \hat{z}_i\alpha$, which covers both dense and sparse with moderate underestimation for common scenarios, and allows column-wise decisions independent of $|C^{\text{UC}}|$ (where sparse-row overheads might be amortized in case of many columns). Columns whose estimated compression ratio $\hat{S}_i^{\text{UC}}/\hat{S}_i^{\text{C}}$ exceeds 1 are added to a compressible set C^{C}. In a last step, we divide the columns in C^{C} into bins and apply the greedy brute-force algorithm within each bin to form column groups.

Compression Phase (lines 13-23)**:** The compression phase first obtains exact information about the parameters of each column group and uses this information in order to adjust the groups, correcting for any errors induced by sampling during planning. The exact information is also used to make the final decision on encoding formats for each group. In detail, for each column group $\mathcal{G}_i$, we extract the "big" (i.e., uncompressed) list that comprises the set $\mathcal{T}_i$ of distinct tuples together with the uncompressed lists of offsets for the tuples. The big lists for all of the column groups are extracted during a single column-wise pass through $\mathbf{X}$ using hashing. During this extraction operation, the parameters d_i, z_i, r_{ij}, and b_{ij} for each group $\mathcal{G}_i$ are computed exactly, with negligible additional cost. These parameters are used in turn to calculate the exact compressed sizes S_i^{RLE} and S_i^{OLE} and exact compression ratio $S_i^{\text{UC}}/S_i^{\text{C}}$ for each group.

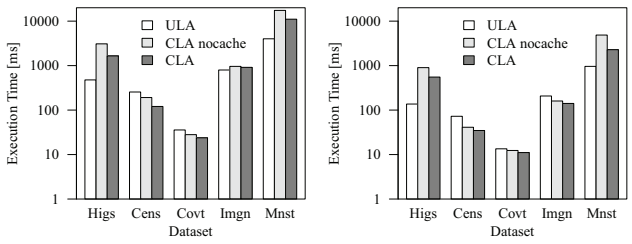
Corrections: Because the column groups are originally formed using compression ratios that are estimated from a sample, there may be false positives, i.e., purportedly compressible groups that are in fact incompressible. Instead of simply storing false-positive OLE/RLE groups as UC group, we attempt to correct the group by removing the column with largest estimated compressed size. The correction process is repeated until the remaining group is either compressible or empty. After each group has been corrected, we choose the optimal encoding scheme for each compressible group $\mathcal{G}_i$ using the exact parameter values d_i, z_i, b_{ij}, and r_{ij} together with the formulas (1) and (2). The incompressible columns are collected into a single UC column group.

5. EXPERIMENTS

We present some highlights from an experimental study of CLA as implemented in SystemML, emphasizing end-to-end results; see [12] for details and additional experiments. Overall, the results show that, for a variety of ML programs and real-world datasets, CLA indeed achieves in-memory matrix-vector multiplication performance close to uncompressed while yielding substantially better compression ratios than lightweight general-purpose compression. As a consequence, CLA provides large end-to-end performance improvements when uncompressed or lightweight-compressed matrices do not fit in local or aggregated memory.

Implementation: When CLA is enabled, SystemML automatically injects—for any multi-column input matrix—a so-called `compress` operator via new rewrites. This applies to both single-node and distributed Spark operations, where the execution type is chosen based on memory estimates. For Spark, we compress individual matrix blocks independently. Making our compressed matrix block a subclass of the uncompressed matrix block yielded seamless integration of all operations, serialization, and buffer-pool interactions.

Experimental Setup: We ran all experiments on a cluster of one head node and six additional nodes; see [12] for details. For our end-to-end experiments, we ran versions of



(a) Single-Threaded (b) Multi-Threaded

Figure 7: Matrix-Vector Multiplication Time.

various ML algorithms from Table 2 over scaled-up versions of the real-world Mnist and ImageNet datasets introduced in Table 1. Specifically, we used the *InfiMNIST* data generator [7] to create an Mnist480m dataset of 480 million observations with 784 features and binomial class labels (1.1 TB in binary format), and also scaled up the ImageNet dataset via replication. We used the ML algorithms in Table 2, with the multinomial variant of logistic regression and the Poisson-regression instantiation of GLM. To isolate the effects of compression, we compare against Apache SystemML 0.9 (Feb 2016) with uncompressed linear algebra (ULA), heavyweight compression (Gzip), and lightweight compression (Snappy); for the latter, we use native compression libraries and ULA. We report end-to-end runtime (average of 3 runs), including read from HDFS, Spark context creation, and compression. The baselines are ULA and Spark's RDD compression with Snappy. In [12], we also compare with CSR-VI [18] and D-VI, a sparse (resp., dense) format with dictionary encoding; our experiments show that CLA has similar operations performance to these algorithms and significantly better compression ratios.

Before describing end-to-end results, we briefly discuss the empirical performance of the compression algorithm and of matrix-vector operations over compressed data.

Compression Speed: Over all datasets, CLA shows reasonably good compression times with a bandwidth ranging from 75.2 MB/s to 121.4 MB/s, single-threaded. Our use of sampling (with the default sampling fraction of 0.01) yielded speedups of up to two orders of magnitude, especially for datasets like Census and Covtype, where a substantial fraction of time is spent on column grouping. In comparison, the single-threaded compression bandwidth of Gzip and Snappy ranged from 6.9 MB/s to 35.6 MB/s and from 156.8 MB/s to 353 MB/s, respectively.

Operations Speed: Figures 7(a) and 7(b) show the single- and multi-threaded matrix-vector multiplication time. Despite row-wise updates of the target vector (which favors uncompressed row-major layout), CLA shows performance close to ULA, with the exceptions of Higgs and Mnist8m, where CLA performs significantly worse. This latter behavior is mostly caused by (1) a large number of values which require multiple passes over the output vector, and (2) the size of the output vector. For Higgs (11M rows) and Mnist8m (8M rows), the target vector does not entirely fit into the L3 cache (15 MB). Accordingly, we see substantial improvements by cache-conscious CLA operations, especially for multi-threaded due to cache thrashing effects. Multi-threaded operations show a speedup similar to ULA due to parallelization over logical row partitions, in some cases even better. Results for vector-matrix multiplication are similar. Overall, we obtain empirical confirmation of our goals for CLA: superior compression ratios (per Table 1) and operation performance comparable to uncompressed. The overall impact on performance is discussed below.

Table 3: Mnist8m Deserialized RDD Storage Size.

Block Size	1,024	2,048	4,096	8,192	16,384
ULA	18 GB	18 GB	18 GB	18 GB	**18 GB**
Snappy	7.4 GB	7.4 GB	7.4 GB	7.4 GB	**7.4 GB**
CLA	9.9 GB	8.4 GB	6 GB	4.4 GB	**3.6 GB**

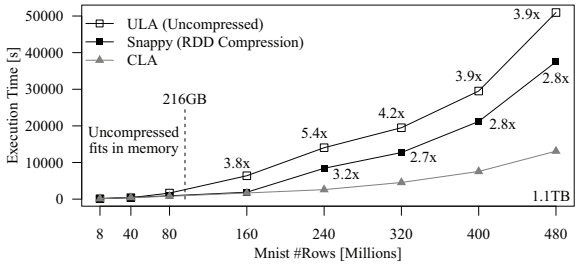


Figure 8: L2SVM End-to-End Performance Mnist.

RDD Storage Size: ULA and CLA use the deserialized storage level `MEM_AND_DISK`, while Snappy uses `MEM_AND_DISK_SER` because RDD compression requires serialized data. ULA also uses `MEM_AND_DISK_SER` for sparse matrices whose sizes exceed aggregated memory. Table 3 shows the RDD storage size of Mnist8m with varying SystemML block size. For 16K, we observe a compression ratio of 2.5x for Snappy but 5x for CLA. We obtained similar ratios for larger Mnist subsets. CLA's compression advantage increases with larger block sizes because the common header is stored only once per column group per block.

L2SVM on Mnist: We first investigate the common classification algorithm L2SVM. An L2SVM model is fit to training data by adjusting its parameters to minimize training error via iterative gradient-descent. For each gradient step, an inner loop searches for the optimal step size. In our setup the aggregated memory size is 216 GB. SystemML uses hybrid runtime plans, where only operations that exceed the driver memory are executed as distributed Spark instructions; all other vector operations are executed—similarly for all baselines—as single-node operations at the driver. For L2SVM, we have two scans of $\mathbf{X}$ per outer iteration (matrix-vector and vector-matrix), whereas all inner-loop operations are purely single-node for the data at hand. Figure 8 shows the results. In reference to our goals from Figure 1, Spark spills data to disk at the granularity of partitions (128 MB as read from HDFS), leading to a graceful performance degradation. As long as the data fits in aggregated memory (Mnist80m, 180 GB), all runtimes are almost identical, with Snappy and CLA showing overheads of up to 25% and 10%, respectively. However, if the ULA format no longer fits in aggregated memory (Mnist160m, 360 GB), we see significant improvements from compression because the size reduction avoids spilling, i.e., reads per iteration. The larger compression ratio of CLA allows to fit larger datasets into memory (e.g., Mnist240m). Once the CLA format no longer fits in memory, the runtime differences converge to the differences in compression ratios.

Other ML Algorithms on Mnist: Next, we study a range of algorithms, including algorithms with RDD operations in nested loops (e.g., GLM, Mlogreg) and non-iterative algorithms (e.g., LinregDS and PCA). Table 4 shows the results for the interesting points of Mnist40m (90 GB), where

Table 4: End-to-End Performance Mnist40m/240m.

Algorithm	**Mnist40m** (90 GB)			**Mnist240m** (540 GB)		
	ULA	Snappy	CLA	ULA	Snappy	CLA
Logreg	630 s	875 s	**622 s**	83,153 s	27,626 s	**4,379 s**
GLM	409 s	647 s	**397 s**	74,301 s	23,717 s	**2,787 s**
LinregCG	**173 s**	220 s	176 s	2,959 s	1,493 s	**902 s**
LinregDS	**187 s**	208 s	247 s	1,984 s	1,444 s	**1,305 s**
PCA	**186 s**	203 s	242 s	1,203 s	**1,020 s**	1,287 s

Table 5: End-to-End Performance ImageNet15/150.

Algorithm	**ImageNet15** (65 GB)			**ImageNet150** (650 GB)		
	ULA	Snappy	CLA	ULA	Snappy	CLA
L2SVM	**157 s**	199 s	159 s	25,572 s	8,993 s	**3,097 s**
Mlogreg	255 s	400 s	**250 s**	100,387 s	31,326 s	**4,190 s**
GLM	190 s	304 s	**186 s**	60,363 s	16,002 s	**2,453 s**
LinregCG	**69 s**	98 s	71 s	3,829 s	997 s	**623 s**
LinregDS	207 s	216 s	**118 s**	3,648 s	2,720 s	**1,154 s**
PCA	211 s	215 s	**119 s**	2,765 s	2,431 s	**1,107 s**

all datasets fit in memory, and Mnist240m (540 GB), where neither uncompressed nor Snappy-compressed datasets entirely fit in memory. For Mnist40m and iterative algorithms, we see similar ULA/CLA performance but a 50% slowdown with Snappy. This is because RDD compression incurs decompression overhead per iteration, whereas CLA's initial compression cost is amortized over multiple iterations. For non-iterative algorithms, CLA is up to 32% slower while Snappy shows less than 12% overhead. Beside the initial compression overhead, CLA also shows less efficient TSMM performance, while the RDD decompression overhead, is mitigated by initial read costs. For Mnist240m, we see significant performance improvements by CLA—of up to 26x and 8x—compared to ULA and RDD compression for Mlogreg and GLM. This is due to many inner iterations with RDD operations in the outer and inner loop. In contrast, for LinregCG, we see only moderate improvements due to a single loop with one matrix-vector chain per iteration, where the CLA runtime was dominated by initial read and compression. Finally, for LinregDS, CLA shows again slightly inferior TSMM performance but moderate improvements compared to ULA. Overall CLA shows positive results with significant improvements for iterative algorithms due to smaller memory bandwidth requirements and reduced I/O.

ML Algorithms on ImageNet: To validate the end-to-end results, we study the same algorithms over replicated ImageNet datasets. Due to block-wise compression, replication did not affect the compression ratio. Table 5 shows the results for ImangeNet15 (65 GB) that fits in memory, and ImageNet150 (650 GB). For LinregDS and PCA, CLA performs better than on Mnist due to superior vector-matrix and thus TSMM performance. Overall, we see similar results with improvements of up to 24x and 7x.

6. CONCLUSIONS

We have shown that compressed linear algebra (CLA)—in which matrices are compressed with lightweight techniques and linear algebra operations are performed directly over the compressed representation—can yield significant performance benefits for common ML algorithms over real-world data. CLA is enabled by declarative ML, which hides the underlying physical data representation. CLA generalizes sparse matrix representations, encoding both dense and sparse matrices in a universal compressed form. CLA is also broadly applicable to any system that provides blocked matrix representations, linear algebra, and physical data independence. Meanwhile, we have also made our CLA prototype available open source in Apache SystemML's 0.11 release. Interesting future work includes (1) full optimizer integration, (2) global planning and physical design tuning, (3) integrating additional compression schemes, and (4) efficient operations beyond matrix-vector.

7. REFERENCES

[1] M. Abadi et al. TensorFlow: A System for Large-Scale Machine Learning. In *OSDI*, 2016.

[2] A. Alexandrov et al. The Stratosphere Platform for Big Data Analytics. *VLDB J.*, 23(6), 2014.

[3] A. Ashari et al. On Optimizing Machine Learning Workloads via Kernel Fusion. In *PPoPP*, 2015.

[4] J. Bergstra et al. Theano: a CPU and GPU Math Expression Compiler. In *SciPy*, 2010.

[5] M. Boehm et al. Declarative Machine Learning – A Classification of Basic Properties and Types. *CoRR*, 2016.

[6] M. Boehm et al. SystemML: Declarative Machine Learning on Spark. *PVLDB*, 9(13), 2016.

[7] L. Bottou. The infinite MNIST dataset. `http://leon.bottou.org/projects/infimnist`.

[8] R. Chitta et al. Approximate Kernel k-means: Solution to Large Scale Kernel Clustering. In *KDD*, 2011.

[9] J. Cohen et al. MAD Skills: New Analysis Practices for Big Data. *PVLDB*, 2(2), 2009.

[10] S. Das et al. Ricardo: Integrating R and Hadoop. In *SIGMOD*, 2010.

[11] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, 2004.

[12] A. Elgohary et al. Compressed Linear Algebra for Large-Scale Machine Learning. *PVLDB*, 9(12), 2016.

[13] I. J. Good. The Population Frequencies of Species and the Estimation of Population Parameters. *Biometrika*, 40(3–4), 1953.

[14] P. J. Haas and L. Stokes. Estimating the Number of Classes in a Finite Population. *J. Amer. Statist. Assoc.*, 93(444), 1998.

[15] B. Huang et al. Cumulon: Optimizing Statistical Data Analysis in the Cloud. In *SIGMOD*, 2013.

[16] B. Huang et al. Resource Elasticity for Large-Scale Machine Learning. In *SIGMOD*, 2015.

[17] N. L. Johnson et al. *Univariate Discrete Distributions*. Wiley, New York, 2nd edition, 1992.

[18] K. Kourtis et al. Optimizing Sparse Matrix-Vector Multiplication Using Index and Value Compression. In *Comput. Frontiers*, 2008.

[19] M. Lichman. UCI Machine Learning Repository: Higgs, Covertype, US Census (1990). `archive.ics.uci.edu/ml/`.

[20] M. Stonebraker et al. The Architecture of SciDB. In *SSDBM*, 2011.

[21] L. Yu et al. Exploiting Matrix Dependency for Efficient Distributed Matrix Computation. In *SIGMOD*, 2015.

[22] M. Zaharia et al. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *NSDI*, 2012.

Updates to the TODS Editorial Board

Christian S. Jensen
csj@cs.aau.dk

It is of paramount importance for a scholarly journal such as *ACM Transactions on Database Systems* to have a strong editorial board of respected, world-class scholars. The editorial board plays a fundamental role in attracting the best submissions, in ensuring insightful and timely handling of submissions, in maintaining the high scientific standards of the journal, and in maintaining the reputation of the journal. Indeed, the journal's associate editors, along with the reviewers and authors they work with, are the primary reason that TODS is a world-class journal.

As of January 1, 2017, three Associate Editors—Divyakant Agrawal, Sihem Amer-Yahia, and Paolo Ciaccia—ended their terms, each having served on the editorial board for roughly six years. In addition, they will stay on until they complete their current loads.

Paolo, Divy, and Sihem have provided very substantial, high-caliber service to the journal and the database community. Specifically, they have lent their extensive experience, deep insight, and sound technical judgment to the journal. I have never seen them compromise on quality when handling submissions. Surely, they have had many other demands on their time, many of which are better paid, during these past six years. We are all fortunate that they have donated their time and unique expertise to the journal and our community during half a dozen years. They deserve our recognition for their commitment to the scientific enterprise.

Also as of January 1, 2017, three new Associate Editors joined the editorial board:

- Feifei Li, University of Utah, `https://www.cs.utah.edu/~lifeifei`
- Kian-Lee Tan, National University of Singapore, `https://www.comp.nus.edu.sg/~tankl`
- Jeffrey Xu Yu, Chinese University of Hong Kong, `http://www.se.cuhk.edu.hk/people/yu.html` c

All three are highly regarded scholars in database systems. We are very fortunate that these outstanding scholars are willing to volunteer their valuable time and indispensable expertise for handling manuscripts for the benefit of our community. Indeed, I am gratified that they have committed to help TODS continue to evolve and improve, and I am looking forward to working with them.