# SIGMOD Officers, Committees, and Awardees


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

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

**Secretary/Treasurer**
Fatma Ozcan
IBM Research
Almaden Research Center
San Jose, California
USA
+1 408 927 2737
fozcan <at> us.ibm.com


**SIGMOD Executive Committee:**
Juliana Freire (Chair), Ihab Francis Ilyas (Vice-Chair), Fatma Ozcan (Treasurer), K. Selçuk Candan, Yanlei Diao, Curtis Dyreson, Yannis Ioannidis, Christian Jensen, and Jan Van den Bussche.

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

**SIGMOD Information Director:**
Curtis Dyreson, Utah State University

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

**SIGMOD Record Editor-in-Chief:**
Yanlei Diao, University of Massachusetts Amherst

**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, Pinar Tözün, 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), Chris Jermaine (co-Chair), Paris Koutris, Feifei Li, Qiong Luo, Ioana Manolescu, Lucian Popa, Renée Miller

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

## SIGMOD Edgar F. Codd Innovations Award

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

Michael Stonebraker (1992)
David DeWitt (1995)
Serge Abiteboul (1998)
Rudolf Bayer (2001)
Ronald Fagin (2004)
Jennifer Widom (2007)
Umeshwar Dayal (2010)
Stefano Ceri (2013)
Gerhard Weikum (2016)
Jim Gray (1993)
C. Mohan (1996)
Hector Garcia-Molina (1999)
Patricia Selinger (2002)
Michael Carey (2005)
Moshe Y. Vardi (2008)
Surajit Chaudhuri (2011)
Martin Kersten (2014)
Goetz Graefe (2017)
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)
Richard Hipp (2017)

## 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)
Yannis E. Ioannidis (2017)
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
- **2017** *Winner*: Peter Bailis. *Honorable Mention*: Immanuel Trummer

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

[Last updated : June 30, 2017]

# Editor's Notes

Welcome to the March 2018 issue of the ACM SIGMOD Record!

The new year of 2018 begins with a special issue on the **2017 ACM SIGMOD Research Highlight Award**. This is an award for the database community to showcase a set of research projects that exemplify core database research. In particular, these projects address an important problem, represent a definitive milestone in solving the problem, and have the potential of significant impact. This award also aims to make the selected works widely known in the database community, to our industry partners, and to the broader ACM community.

The award committee and editorial board included Zack Ives, Jeff Naughton, Wang-Chiew Tan, and Yanlei Diao. We solicited articles from PODS 2017, SIGMOD 2017, VLDB 2017, ICDE 2017, EDBT 2017, and ICDT 2017, as well as from community nominations. Through a careful review process five articles were finally selected as 2017 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 the topic presented in the article. The technical perspective provides the reader with an overview of the background, the motivation, and the key innovation of the featured research highlight, as well as its scientific and practical significance.

The 2017 research highlights cover a broad set of topics, including (a) a new theoretical framework for feature engineering for programming machine-learning solutions over a database ("A Relational Framework for Classifier Engineering"); (b) a parallel graph processing system that employs a simple, intuitive programming model and a principled approach based on fixpoint computation which enables database-style optimization ("From Think Parallel to Think Sequential"); (c) a scalable linear algebra system built on top of a parallel relational database system ("Scalable Linear Algebra on a Relational Database System"); (d) an entity matching system that overcomes limitations of existing solutions by considering the requirements for building an end-to-end system ("Magellan: Toward Building Entity Matching Management Systems"); (e) a new approach to helping the user understand answers of natural language queries, i.e., giving an explanation of how and why each answer exists ("Natural Language Explanations for Query Results").

On behalf of the SIGMOD Record Editorial Board, I hope that you enjoy reading the March 2018 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 2018

## Past SIGMOD Record Editors:

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

# Technical Perspective: A Relational Framework for Classifier Engineering

Wang-Chiew Tan
Recruit Institute of Technology
wangchiew@recruit.ai

A fundamental step in developing machine-learning solutions is that of *feature engineering*. Feature engineering refers to the process of generating a representation from data (called *features*) that can be fed as inputs to machine-learning models. The results of feature engineering thus have direct impact on the performance of machine-learning models. In developing machine-learning solutions, a large amount of time is typically devoted to feature engineering, which determines the right features to capture for improving the performance of the models.

In this paper, the authors describe a framework for feature engineering for programming machine-learning solutions over a database, assuming the model inputs numerical paramaters that may be tuned by fitting to training examples. The focus of the paper is on a widely used class of machine-learning models, called *classifiers*, which are used to predict an unknown category of a given entity based on the properties of that entity.

The running example of the paper considers the problem where a credit card company wishes to identify whether an incoming credit card transaction is a legitimate or fraudulent transaction (e.g., made with a stolen credit card). The credit card company may leverage historical transactions with both legitimate and fraudulent transactions as training data to train a classifier. The features that are extracted from the data may include properties that concern the state and country where a transaction was made compared to the state and country of billing address of the owner, the amount billed in the transaction, the history of transactions and so on.

Their framework assumes an underlying *entity schema*, which is a relation schema with a distinguished relation symbol. The distinguished relation represents the set real-world objects where the classifier makes predications upon. For the credit card example, since the classifier will ultimately be applied on transactions to determine the legitimacy of transactions, a natural candidate for the distinguished entity relation in the credit card example is the transaction relation which stores all transactions that occurred. The remaining relation schema will include additional information about the transaction, such as the country and state where each transaction took place, the card and amount involved, and information about the billing address of the credit card. Feature engineering is modeled as the process where an analyst specifies a sequence of *feature queries* in some language. For example, a feature query may select all transactions that took place in the same country and state of the owner's billing address, and another feature query may select all the ones that took place in the same country (but not necessarily the same state) of the owner. In their framework, a classifier is a function that maps a vector of numbers, where the numbers encode the features, into a boolean answer. To train a classifer, it is therefore necessary to convert the results of feature queries into numbers. This is done as follows. For every feature query and every entity in the set of entities (in this case, transactions), a vector of 1 or -1 can be obtained based on whether the feature query produces an answer for that entity. If an answer is produced (resp. not produced), then the feature query for that transaction is a positive example which will be given a label 1 (resp. negative example labeled -1). Based on the input vectors obtained this way, a classifier is learnt and can then be applied to future inputs.

After formalizing a relational framework for feature engineering, the authors further describe three fundamental problems. The *separabiity* problem essentially asks the question that for a given language of the framework and a given category of classifiers, whether it is sufficient to achieve good classification (i.e., separation of the positive from the negatives) based on the training data. More precisely, given a training instance over an entity schema, determine whether or not there exists a sequence of feature queries and a classifier that completely agrees with the training labels. Another problem, called the Vapnik-Chervonenkis dimensionality (*VC dimensionality*) problem, essentially asks what is the complexity of learnability. More precisely, the VC dimenion measures the complexity of a class of classifiers (e.g., linear or polynomial classifiers) for feature queries specified in some language. The last problem posed by the authors is the *identifiability* problem. This problem asks to decide, given a sequence of feature queries, whether there exists a training instance such that the resulting feature matrix is of full column dimension. In other words, the columns of the matrix are linearly independent.

The paper presents complexity results on these three problems with attention on linear classifiers and where feature queries are conjunctive queries. Perhaps more importantly, the paper presents a novel formal framework for classifier engineering, describes an initial set of results based on the framework, and leaves several open challenges for the interested reader.

# A Relational Framework for Classifier Engineering

Benny Kimelfeld
Technion – Israel Institute of Technology
Haifa 32000, Israel
bennyk@cs.technion.ac.il

Christopher Ré
Stanford University
Stanford, CA 94305, USA
chrismre@cs.stanford.edu

## ABSTRACT

In the design of analytical procedures and machine-learning solutions, a critical and time-consuming task is that of feature engineering, for which various recipes and tooling approaches have been developed. We embark on the establishment of database foundations for feature engineering. Specifically, we propose a formal framework for classification in the context of a relational database. The goal of this framework is to open the way to research and techniques to assist developers with the task of feature engineering by utilizing the database's modeling and understanding of data and queries, and by deploying the well studied principles of database management. We demonstrate the usefulness of the framework by formally defining key algorithmic challenges and presenting preliminary complexity results.

## 1. INTRODUCTION

A critical and time-consuming task in the development of analytics and machine-learning solutions is that of *feature engineering* [21,35]. Given its importance, recipes and tooling have been developed for practitioners [1, 18]. With the advent of frameworks like SAS, Cloudera's IBIS and Oracle's ORE, feature engineering is often carried out over relational data. Thus, a pressing challenge is to understand how to merge these analytics with traditional database management techniques. To this end, we propose a relational framework for *classification*—a simple and popular analytic task.

The task of feature engineering is that of generating inputs (or *signals*) from available data, in order to improve the performance of the underlying model in solving a target problem. This target problem is typically classification (predicting an unknown category of a given entity), or regression (predicting the value of an unknown function on a given entity). The model makes its prediction based on various properties, called *features*, of the given entity. We focus here on *parametric models*, where the model has a pre-determined structure with numerical parameters that are tuned by fitting to training examples, a process termed *learning*.

This article is a minor revision of the work published in PODS 2017, May 14-19, 2017, Raleigh, NC, USA, ©2017 ACM. ISBN978-1-4503-4198-1/17/05...$15.00 DOI: http://dx.doi.org/10.1145/3034786.3034797



Naturally, the choice of features has a major impact on the resulting model. A suboptimal set of features may lead to *overfitting* (where the learned model does not well generalize beyond the training examples), or to *underfitting* (the model is incapable of capturing the target function due to lack of information or expressiveness) [18]. Another consideration is that of *execution cost*, as costly features may result in an impractical model [35]. There are also legal and moral considerations in cases where decision makers are required to practice fairness and lack of discrimination; there, a central challenge is to select appropriate features [10, 34, 36].

Our running example is a scenario where the security branch of a credit-card company aims to make an educated guess on whether an incoming transaction is a fraudulent purchase (e.g., made on a stolen card). To make such a guess, the company uses information available in its database, such as whether there were similar transactions in the past, whether the purchase is made in the same state of the owner's mailing address, or in the same country, the amount being paid, and so on. An employed engineer then selects a machine-learning library and learns a classifier, which can be trained based on past data on fraudulent activity.

The classifier is simply a function that maps a vector of numbers into a yes/no decision; in turn, these numbers, called *features*, encode relevant pieces of information (e.g., $f = +1/-1$ depending on whether or not the transaction is in the country of the owner). The machine-learning library typically tunes parameters (or weights) of the classifier by fitting them to the examples. In our scenario, high quality is crucial: false positives disrupt legitimate business, and false negatives cause financial losses and customer distress. So, the engineer produces additional features to consider by phrasing different questions (*feature queries*) about the transaction at stake, until she is satisfied with the results. This activity is referred to as *feature engineering*.

Analysts typically spend the bulk of their time on feature engineering [17,21,35]. This process includes trial and error, and stepwise addition or removal of features [18]. Involving the database semantics in feature engineering has the potential to automate some important tasks, and thereby assist the engineer. For instance, the engineer may ask whether some class of simple feature queries (e.g., select-project-join) suffices to achieve good classification based on the training data, or otherwise more expressiveness is needed. This motivates the *separability* problem that we later discuss. She may also ask the complementary question, which is whether the current set of feature queries is too detailed and allows the model to *overfit* the examples and poorly generalize to

future transactions. A traditional way to measure overfitting potential is via the *Vapnik-Chervonenkis* (*VC*) dimension [31]. The features of choice may have impact on the VC dimension, and we later refer to the problem of determining this impact as *VC dimensionality*.

The engineer may also encounter the common technical challenge where the machine-learning library fails due to incompatible data, as it requires the matrix of training data to be of a full (column) degree. Partial degree (column dependence) may be an artifact of the training examples; but it may also be an inherent error in the feature design. As an example, for US owners the features "payment is in the US," "payment is in the owner's state," and "payment is in a different US state" have an inherent linear dependence among them: the first minus the second equals the third. We could use the database to detect such problems. This leads to the problem that we later refer to as *identifiability*.

### Contribution

Our goal is to establish the first steps in a database theory that embeds the automation of core tasks in feature selection. More precisely, our framework aims to open the way to novel research and techniques for utilizing the database's understanding of raw data and queries, in order to fundamentally assist with the process of feature engineering.

The framework is based on an *entity schema*, which is a relational schema with a distinguished relation symbol that represents *entities*. A *database instance* over an entity schema represents a collection of entities, along with additional (direct or indirect) knowledge about these entities. A *feature query* selects entities with a certain property, and a *statistic* is a sequence of feature queries. The central goal in classification is to train and apply a *classifier* (or a *classifier model*), which is a function that takes as input the *statistic* of an entity (i.e., the vector obtained by applying each feature query) and outputs a $+1/-1$ decision. In *training* a classifier, we are given a set of entities labeled with $+1/-1$, and produce a classifier from a predefined *model class*. A linear classifier, for example, is encoded as a vector of weights over the features in the statistic. In our framework, the collection of training examples is represented simply by an instance over the entity schema, along with a labeling function that maps each entity to $+1/-1$.

For illustration, Figure 1(a) shows an entity schema $\mathbf{S}$ where entities are transactions (identified by numbers), and Figure 1(b) shows a statistic $\Pi$ over $\mathbf{S}$ with two feature queries $\pi_1$ and $\pi_2$. Figure 2 shows the training workflow in our framework: an instance $I$ over $\mathbf{S}$ with a labeling of the entities (transactions) is transformed into a $(+1/-1)$-matrix, with a feature row per entity, and the matrix is used for building a classifier $h$. This classifier predicts the labels in a new instance, as illustrated in Figure 3.

The features, as defined above, are based on boolean properties of the entities: if the entity satisfies the property (i.e., it is "selected"), then the feature value is $+1$, and otherwise $-1$. Boolean features are highly important in practice, and in fact, we are aware of quite a few deployments where numerical values are translated into boolean ones (e.g., by means of *bucketing*, or *binning*, numbers into intervals). The more general case is that in which the feature query associates a numerical value with each entity, and we discuss this generalization as a future direction later on.

Our framework enables to formalize relevant computational problems, analyze their complexity, and ultimately design algorithmic solutions. Here, we formally define three computational problems that capture tasks in the construction and evaluation of features: *separability*—whether a perfect separator exists for a training set; *VC dimensionality*—computing the fundamental measure of the complexity of a classifier class; and *identifiability*—testing for a statistical property guaranteeing that the classifier model is uniquely defined given enough data.

Our analysis focuses on linear classifiers and features definable as conjunctive queries. Specifically, we show a tight relationship between the computational complexity of our problems and that of *query containment* (and *equivalence*): it is necessary, and often sufficient, to solve containment in order to solve our problems. In this article, we present our complexity results on the separability problem. Additional results and proofs on all three problems can be found in the conference version of this article [22].

## 2. PRELIMINARIES

In this section we give the basic definitions and terminology that we use throughout the article.

### Relational Databases

Our relational terminology is as follows. A *schema* is a collection of *relation symbols*. Each relation symbol $R$ has an associated *arity* $k$. We assume an infinite set $\mathsf{Const}$ of *constants*. An *instance* $I$ over a schema $\mathbf{S}$ associates with every $k$-ary relation symbol $R \in \mathbf{S}$ a finite subset of $\mathsf{Const}^k$. We denote by $R^I$ the relation that $I$ associates with the relation symbol $R$. The *active domain* of an instance $I$, denoted $adom(I)$, is the set of all the constants in $\mathsf{Const}$ that are mentioned in the tuples of $I$. A *fact* over a schema $\mathbf{S}$ is an expression of the form $R(c_1, \ldots, c_k)$, where $R$ is a $k$-ary relation symbol and $c_1, \ldots, c_k$ are constants. We say that the fact $R(c_1, \ldots, c_k)$ *belongs to* an instance $I$ over $\mathbf{S}$ if $R^I$ contains the tuple $(c_1, \ldots, c_k)$. For convenience, we view an instance as the set of its facts. In particular, by $f \in I$ we denote that $f$ belongs to $I$.

Comment 2.1. While schema *constraints* are important in our framework, they are excluded from the basic framework for simplicity sake. Moreover, the complexity results we give later on (Section 5) are oblivious to such constraints. We further discuss constraints in Section 7. □

Let $I$ and $J$ be two instances over the same schema $\mathbf{S}$. A *homomorphism* from $I$ to $J$ is a mapping $\mu : adom(I) \to adom(J)$ such that for every fact $f \in I$ we have $\mu(f) \in J$; here, $\mu(f)$ is the fact that is obtained from $f$ by replacing each constant $a$ with the constant $\mu(a)$.

### Queries

A *query* over a schema $\mathbf{S}$ is a function $q$ that is associated with an arity $k$, and that maps every instance $I$ over $\mathbf{S}$ into a finite subset $q(I)$ of $\mathsf{Const}^k$. A query $q'$ *contains* a query $q$, in notation $q \subseteq q'$, if $q(I) \subseteq q'(I)$ for all instances $I$ over $\mathbf{S}$. If $q \subseteq q'$ and $q' \subseteq q$ then $q$ and $q'$ are said to be *equivalent*. We have a special interest in *unary* queries $q$ (i.e., where $k = 1$); then, by a slight abuse the notation, we view $q(I)$ as a set of constants $a$ rather than a set of tuples $(a)$.

We consider conjunctive queries *without constants*. Formally, a Conjunctive Query (CQ) over a schema $\mathbf{S}$ is a logical

**Table 1: Main symbols**

| | |
|---|---|
| $h$ | hypothesis/classifier $\{-1,1\}^n \to \{-1,1\}$ |
| $\mathbf{H}$ | hypothesis class |
| $\mathsf{Lin}$ | the class of linear classifiers |
| $\mathbf{S}$ | relational/entity schema |
| $\eta$, $\eta_{\mathbf{S}}$ | entity relation (unary) |
| $I$, $J$ | database instance |
| $\eta^I$, $\eta^I_{\mathbf{S}}$ | entity set of the instance $I$ |
| $e$ | entity in $\eta^I_{\mathbf{S}}$ |
| $\mathbf{QL}$ | query language |
| $\mathsf{CQ}$ | the class of CQs (without constants) |
| $\pi$ | feature query (unary) |
| $\pi^I(e)$ | $+1$ if $e \in \pi(I)$ and $-1$ if $e \notin \pi(I)$ |
| $\Pi$ | statistic $(\pi_1, \ldots, \pi_n)$ |
| $\lambda$ | labeling function $\eta^I_{\mathbf{S}} \to \{-1,1\}$ |

formula $q(\mathbf{x})$ of the form

$$\exists \mathbf{y}[\phi_1(\mathbf{x},\mathbf{y}) \wedge \cdots \wedge \phi_m(\mathbf{x},\mathbf{y})]$$

where $\mathbf{x}$ and $\mathbf{y}$ are disjoint sequences of variables and each $\phi_i$ is an atomic query over $\mathbf{S}$ (i.e., a formula that consists of a single relation symbol and no logical operators) without constants. Observe that in this article CQs do not contain built-in relations such as $x > y$; we use this assumption in our analysis. The atomic formula $\phi_i$ is called an *atom* of $q$. We use the conventional notation

$$q(\mathbf{x}) \leftarrow \phi_1(\mathbf{x},\mathbf{y}), \cdots, \phi_m(\mathbf{x},\mathbf{y})$$

to denote a CQ. The left side $q(\mathbf{x})$ is called the *head* and the right side $\phi_1(\mathbf{x},\mathbf{y}), \cdots, \phi_m(\mathbf{x},\mathbf{y})$ is called the *body*. We require each variable in the head to occur at least once in the body. We may refer to a CQ by mentioning only its head $q(\mathbf{x})$ or even just $q$. We denote by $\mathsf{CQ}$ the class of CQs (as defined here, i.e., without constants).

Let $\mathbf{S}$ be a schema, let $q$ be a CQ over $\mathbf{S}$, and let $I$ be an instance over $\mathbf{S}$. A *homomorphism* from $q$ to $I$ is a mapping from the variables of $q$ to $adom(I)$, such that for every atom $\phi$ of $q$, the fact $\mu(\phi)$ belongs to $I$; here, $\mu(\phi)$ is the fact that is obtained from $\phi$ by replacing each variable $z$ with the constant $\mu(z)$. The result of applying the CQ $q(\mathbf{x})$ to the instance $I$ is the relation that consists of all the tuples $\mu(\mathbf{x})$, where $\mu$ is a homomorphism from $q$ to $I$ and $\mu(\mathbf{x})$ is obtained from $\mathbf{x}$ by replacing every variable $x_i$ with $\mu(x_i)$. We denote this relation by $q(I)$.

### *Classifiers and Learning*

In this work, a *classifier* is a function of the form

$$h : \{-1,1\}^n \to \{-1,1\}$$

where $n$ is a natural number that we call the *arity* of $h$. A *hypothesis class* is a (possibly infinite) family $\mathbf{H}$ of classifiers, and a classifier in $\mathbf{H}$ is referred to as a *hypothesis*. We denote by $\mathbf{H}_n$ the restriction of $\mathbf{H}$ to the $n$-ary hypotheses in $\mathbf{H}$. An $n$-ary *training collection* is a multiset $T$ of pairs $\langle \mathbf{x}, y\rangle$ where $\mathbf{x} \in \{-1,1\}^n$ and $y \in \{-1,1\}$. We denote by $\mathbf{T}_n$ the class of all $n$-ary training collections. A *cost function* for a hypothesis class $\mathbf{H}$ is a function of the form

$$c : \big(\cup_n (\mathbf{H}_n \times \mathbf{T}_n)\big) \to \mathbb{R}_{\geq 0}$$

where $\mathbb{R}_{\geq 0}$ is the set of nonnegative numbers. Given a training collection $T$ and two hypotheses $h_1$ and $h_2$, the inequality $c(h_1,T) > c(h_2,T)$ implies that $h_2$ is preferred to $h_1$ according to $c$. In the context of a fixed hypothesis class $\mathbf{H}$ and a cost function $c$, *learning* a classifier is the task of finding a hypothesis $h \in \mathbf{H}_n$ that minimizes $c(h,T)$, given a training collection $T \in \mathbf{T}_n$.

It is importnat to allow $T$ to be a multiset in order to enable the scoring function to account for the frequency (rather than pure existence) of examples. For the scope of this article, though, being a multiset does not play any role, and the reader may view $T$ simply as a set.

We illustrate our definitions on the important class of *linear classifiers*. An $n$-ary linear classifier is parameterized by a vector $\mathbf{w} = (w_0, \ldots, w_n) \in \mathbb{R}^{n+1}$, denoted by $\Lambda_{\mathbf{w}}$, and defined as follows for all $\mathbf{a} \in \{-1,1\}^n$.

$$\Lambda_{\mathbf{w}}(\mathbf{a}) \stackrel{\text{def}}{=} \begin{cases} 1 & \text{if } \mathbf{a}\cdot\mathbf{w}' \geq w_0; \\ -1 & \text{otherwise.} \end{cases}$$

where $\mathbf{w}' = (w_1, \ldots, w_n)$ and "$\cdot$" denotes the operation of dot product. By $\mathsf{Lin}$ we denote the class of linear classifiers. An example of a cost function is the *least square* cost that is given by

$$lsq(\Lambda_{\mathbf{w}}, T) \stackrel{\text{def}}{=} \sum_{\langle \mathbf{x}, y\rangle \in T} \big(\mathbf{x}\cdot\mathbf{w}' - w_0 - y\big)^2$$

for the arguments $\Lambda_{\mathbf{w}} \in \mathsf{Lin}_n$ and $T \in \mathbf{T}_n$.

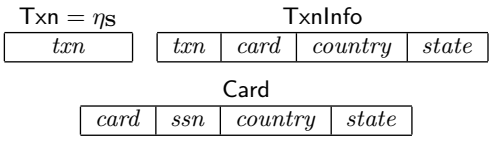
More background on the basic theory of machine-learning classifiers, as well as the relevant linear algebra discussed in the next section, can be found in standard machine-learning textbooks, such as Shalev-Shwartz and Ben-David [28].

**Matrix independence.** We denote by $\mathbf{0}^n$ the vector of $n$ zeroes, and by $\mathbf{1}^n$ the vector of $n$ ones. Let $M$ be an $n \times m$ real matrix (consisting of $n$ rows and $m$ columns). A *linear column dependence* in $M$ is a vector $\mathbf{w} \in \mathbb{R}^m$ such that $\mathbf{w} \neq \mathbf{0}^m$ and $M \cdot \mathbf{w} = \mathbf{0}^n$. A linear column dependence $\mathbf{w}$ in $M$ is an *affine dependence* in $M$ if $\mathbf{w} \cdot \mathbf{1}^m = 0$ (i.e., the components of $\mathbf{w}$ sum up to 0). If $M$ does not have any linear column dependence, then we say that $M$ is *linearly column independent*. Similarly, if $M$ does not have any affine column dependence, then we say that $M$ is *affinely column independent*. Note that linear independence implies affine independence, but the other direction is not necessarily true.

## 3. FRAMEWORK

We now present our formal framework. A basic notion in this framework is that of an *entity schema*, which is simply an ordinary relational schema with a distinguished relation symbol for representing *entities*. For simplicity, we assume that an entity is represented by a single constant (an identifier), hence the corresponding relation is unary. Formally, an entity schema is a pair $(\mathbf{S}, \eta)$, where $\mathbf{S}$ is a schema and $\eta$ is a unary relation symbol in $\mathbf{S}$. An *instance* over an entity schema $(\mathbf{S}, \eta)$ is simply an instance over $\mathbf{S}$. In the remainder of this article all the schemas we consider are entity schemas. So, to simplify the presentation we refer to the entity schema $(\mathbf{S}, \eta)$ simply as $\mathbf{S}$, and refer to $\eta$ as $\eta_{\mathbf{S}}$.

Let $I$ be an instance over an entity schema $\mathbf{S}$. An *entity* of $I$ is a constant $a$ such that $\eta_{\mathbf{S}}(a) \in I$. Hence, $I$ represents a set of entities along with information about the entities. This information is contained in the remaining re-




(a) An entity schema $\mathbf{S}$

$$\pi_1(t) \leftarrow \mathsf{TxnInfo}(t, n, c, s), \mathsf{Card}(n, i, c, s)$$
$$\pi_2(t) \leftarrow \mathsf{TxnInfo}(t, n, c, s), \mathsf{Card}(n, i, c, s')$$

(b) A statistic $\Pi = (\pi_1, \pi_2)$ over $\mathbf{S}$

**Figure 1: An entity schema and a statistic**

lations, which can be joined with $\eta_{\mathbf{S}}$. Again, by a slight abuse of notation, we treat $\eta_{\mathbf{S}}^I$ as the set of all entities of $I$. For example, $e \in \eta_{\mathbf{S}}^I$ means that $e$ is an entity of $I$. A *feature query* (*over* $\mathbf{S}$) is a unary query $\pi$ over the schema $\mathbf{S}$. When the feature query $\pi$ is represented in a query language $\mathbf{QL}$ (e.g., CQ), we say that $\pi$ is *in* $\mathbf{QL}$.

Example 3.1. We use a running example that instantiates the credit-card scenario from the Introduction. Figure 1(a) depicts the entity schema $\mathbf{S}$ with a unary relation Txn, which is $\eta_{\mathbf{S}}$, and two quaternary relations TxnInfo and Card. The box on the top of Figure 3 depicts an instance $I'$. The entities are the transaction identifiers 5, 6 and 7, and these are the members of $\eta_{\mathbf{S}}^{I'} = \{5, 6, 7\}$. Figure 1(b) shows two feature queries in CQ: the feature $\pi_1$ selects all transactions that took place in the same country and state of the owner's maling address, and the feature query $\pi_2$ selects all the ones that took place in the same country (but not necessarily the same state) of the owner. Indeed, in $\pi_1$ the two atoms use the same variable, $s$, for the state, while in $\pi_2$ the first atom uses $s$ and the second uses $s'$. □

Let $I$ be an instance over an entity schema $\mathbf{S}$, and let $\pi$ be a feature query. We define the function $\pi^I : \eta_{\mathbf{S}}^I \to \{-1, 1\}$ as follows.

$$\pi^I(e) = \begin{cases} 1 & \text{if } e \in \pi(I); \\ -1 & \text{otherwise.} \end{cases}$$

Let $\mathbf{S}$ be an entity schema. A *statistic* (over $\mathbf{S}$) is a sequence $\Pi = (\pi_1, \ldots, \pi_n)$ of feature queries. We say that $\Pi$ is in a query language $\mathbf{QL}$ if each $\pi_i$ is in $\mathbf{QL}$. Given an instance $I$ over $\mathbf{S}$, we denote by $\Pi^I$ the function $(\pi_1^I, \ldots, \pi_n^I)$ from $\eta_{\mathbf{S}}^I$ to $\{-1, 1\}^n$ that maps every entity $e \in \eta_{\mathbf{S}}^I$ to the sequence $(\pi_1^I(e), \ldots, \pi_n^I(e))$.

Example 3.2. Figure 1(b) describes the statistic $\Pi = (\pi_1, \pi_2)$ over the schema $\mathbf{S}$ of Figure 1(a). The middle layer of Figure 3 contains (on its left) the tuples $\Pi^{I'}(e)$ for the entities $e$ in the instance $I'$ of the top box in this figure. For example, the top row corresponds to $\Pi^{I'}(5) = (1, 1)$, which is due to the fact Transaction 5 took place in the same country and state of the card holder. □

Let $\mathbf{S}$ be an entity schema. A *labeling* of an instance $I$ over $\mathbf{S}$ is a function

$$\lambda : \eta_{\mathbf{S}}^I \to \{-1, 1\}$$

that partitions the entities into *negative examples* (i.e., entities $e$ where $\lambda(e) = -1$) and *positive examples* (i.e., entities $e$ where $\lambda(e) = 1$). A *training instance* over $\mathbf{S}$ is a pair $(I, \lambda)$, where $I$ is an instance over $\mathbf{S}$ and $\lambda$ is a labeling of $I$. Taken together, a statistic $\Pi$ and a training instance define a training collection, namely, the one that consists of the tuple $\langle \Pi^I(e), \lambda(e) \rangle$ for every entity $e \in \eta_{\mathbf{S}}^I$.

Example 3.3. Continuing our running example, Figure 2 depicts a training instance $(I, \lambda)$ over the entity schema $\mathbf{S}$ of Figure 1(a), where $\lambda$ is represented in the Txn relation. With the statistic $\Pi$ of Figure 1(b) we get the training collection in the left bottom part of Figure 2. From this training instance a classifier $h$ is learned, and is applied for prediction on future instances, as illustrated in Figure 3 for the instance $I'$ that we referred to in the previous examples. □

# 4. COMPUTATIONAL PROBLEMS

We now define three computational problems that are motivated by the design of machine-learning solutions, and feature engineering in particular.

## 4.1 Separability

Separability is perhaps the most basic notion of learning. The traditional presentation of learning theory typically begins with the "noise free" case where the labeled examples are required to be perfectly separated by the features. In our framework, *separability* refers to the following task: given a training instance over an entity schema, determine whether there exists a statistic and a classifier that agree with (i.e., classify precisely as) the example labels. Separability is a simplification of the more general problem, where some noise is allowed (and say, 99% of the examples are required to be correctly satisfied). We adopt the simplified (textbook) task as a first step, and show that it already leads to nontrivial insights within our framework.

The problem is parameterized by two important components: the family of classifiers in consideration, and the query language used for phrasing feature queries. The formal defintion of the problem is as follows.

Let $\mathbf{S}$ be a schema, $\Pi$ a statistic over $\mathbf{S}$, and $\mathbf{H}$ a hypothesis class. A training instance $(I, \lambda)$ is $\mathbf{H}$-*separable* with respect to (w.r.t.) $\Pi$ if there exists a hypothesis $h \in \mathbf{H}$ that *fully agrees* with $\lambda$, that is, $h$ and $\Pi$ have the same arity and $h(\Pi^I(e)) = \lambda(e)$ for every $e \in \eta_{\mathbf{S}}^I$.

Problem 1 (Separability). *For a hypothesis class* $\mathbf{H}$ *and a query language* $\mathbf{QL}$*, the problem* $(\mathbf{H}, \mathbf{QL})$-separability *is the following. Given an entity schema* $\mathbf{S}$ *and a training instance* $(I, \lambda)$ *over* $\mathbf{S}$*, determine whether there exists a statistic* $\Pi$ *in* $\mathbf{QL}$ *such that* $(I, \lambda)$ *is* $\mathbf{H}$*-separable w.r.t.* $\Pi$.

Example 4.1. We continue with our running example, and consider the training instance $(I, \lambda)$ of Figure 2. Suppose that $\mathbf{H}$ is the class Lin of linear classifiers, and that $\mathbf{QL}$ is the class CQ. Then $(I, \lambda)$ is a "yes" instance of the separability problem, and a witness is the statistic $\Pi = (\pi_1, \pi_2)$ of Figure 1(b) with the classifier $\pi_2 - \pi_1 \geq 1$. Now suppose that we add an entity 5, the tuple (5, 102, US, AL) to TxnInfo, and the labeling $\lambda(5) = -1$. The new training instance then becomes a "no" instance of the separability problem since, intuitively, there is no way to distinguish between 4 and 5 using $\mathbf{QL}$ over $I$, and yet, $\lambda$ labels 4 and 5 differently. □

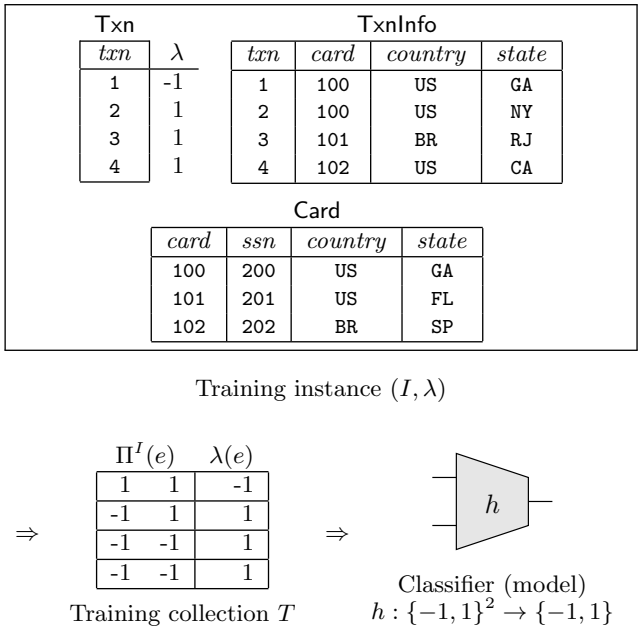



**Figure 2: The training process**

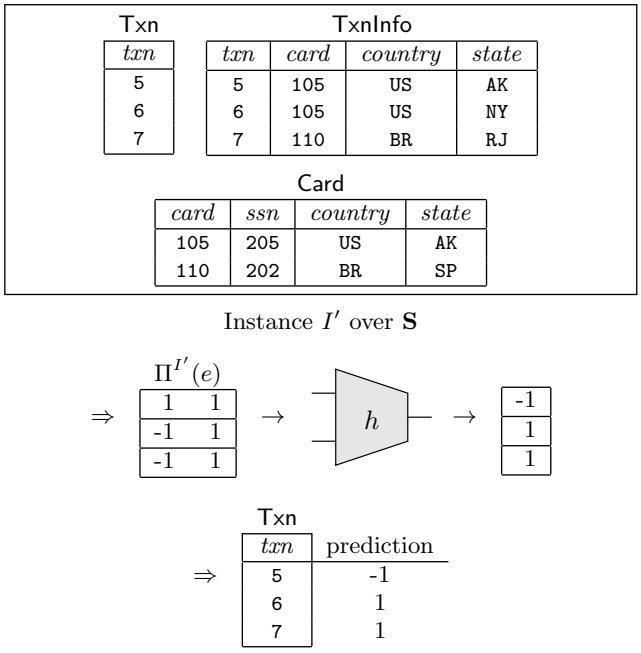



**Figure 3: The prediction process**

## 4.2 VC Dimensionality

The *Vapnik-Chervonenkis* (*VC*) dimension [31] is a measure of complexity of a hypothesis class, and is a de facto complexity measure for learnability. Bounds for *generalization* (how well a learned classifier does on unseen data) typically depend on the VC dimension. It measures the capacity of the classifier class, and is a key indicator of how much data one needs to reliably train the classifier: if this amount is low with respect to the VC dimension, then the classifier may overfit. If the amount of training data is high with respect to the VC dimension, we may be missing opportunities to devise a more accurate classifier.

As an example, the class of *polynomial* classifiers is more expressive than that of the *linear* classifiers, so there is a higher capability of a polynomial-classifier learner to overfit, that is, exploit properties that are exhibited scarcely in the training examples but are not representative of the general population. Similarly, a *deep* decision tree might be constructed to handle every individual example, while a *shallow* one will have to utilize common properties, and hence, intuitively, to better generalize. VC dimension is a mathematical measure that aims to capture this expressive power in a manner that is uniform across model classes. Higher VC dimension implies a more complicated classifier space with higher ability to overfit training data, and so, more training data is required for effective learning.

In our framework, VC dimension is a function of not only the hypothesis class, but also the statistic that translates entities into feature vectors. The formal definition follows.

Let $\mathbf{S}$ be a schema, $\Pi$ a statistic over $\mathbf{S}$, and $\mathbf{H}$ a hypothesis class. An instance $I$ over $\mathbf{S}$ is *shattered by* $\mathbf{H}$ *w.r.t.* $\Pi$ if for every labeling $\lambda$ of $I$ there exists a hypothesis $h \in \mathbf{H}$ that fully agrees with $\lambda$. The *VC dimension of* $\mathbf{H}$ *w.r.t.* $\Pi$ is the maximal number $m$ such that there is an instance $I$ over $\mathbf{S}$ where $I$ has $m$ entities and $I$ is shattered by $\mathbf{H}$ w.r.t. $\Pi$.

Problem 2 (Dimensionality). *Let* $\mathbf{H}$ *be a hypothesis class and* $\mathbf{QL}$ *a query language. The computational problem* $(\mathbf{H}, \mathbf{QL})$*-dimensionality is the following. Given an entity schema* $\mathbf{S}$ *and a statistic* $\Pi$ *in* $\mathbf{QL}$*, compute the VC dimension of* $\mathbf{H}$ *w.r.t.* $\Pi$.

Example 4.2. Recall $\mathbf{S}$ and $\Pi$ of our running example (Figure 1). Computing the VC dimension of $\mathsf{Lin}$ w.r.t. $\Pi$ is an instance of $(\mathsf{Lin}, \mathsf{CQ})$-dimensionality. Our results [22] imply that this dimension is 3. Hence, there exists an instance $I$ with three entities, such that we can find a perfect linear classifier for every labeling $\lambda$ for $I$. Yet, no such instance exists with four or more entities. In this example, then, modeling of the features as CQs does not reduce the VC dimension compared to traditional machine learning where one can freely set the feature values. □

## 4.3 Statistic Identifiability

*Identifiability* asks whether it is possible for one to learn the parameters of the given classifier model *unambiguously* from *some* data set. Here, the question refers to a given statistic, and we consider the case where training is done by means of optimization via linear algebra; we ask whether the space of solutions is bounded. More formally, this problem boils down to deciding, given a statistic, whether there exists any training instance such that the resulting feature matrix is of full column dimension (i.e., the columns are linearly independent). We also consider the variant where linear independence is relaxed to affine independence (as defined in Section 2.) For additional background on identifiability, we refer the reader to Wainwright and Jordan's survey [32]. Next, we give the formal definition.

Let $\mathbf{S}$ be an entity schema, $\Pi$ a statistic over $\mathbf{S}$, and $I$ an instance over $\mathbf{S}$. We fix an arbitrary order over the entities of $I$, and denote by $[\![\Pi^I]\!]$ the matrix that consists of the rows $\Pi^I(e)$ for every $e \in \eta^I_{\mathbf{S}}$ in order. We say that $\Pi$ is *linearly identifiable* if there exists an instance $I$ over $\mathbf{S}$ such that the matrix $[\![\Pi^I]\!]$ is linearly column independent. We say that $\Pi$ is *affinely identifiable* if there exists an instance $I$ over $\mathbf{S}$ such

that the matrix $[\![\Pi^I]\!]$ is affinely column independent. Note that whenever $\Pi$ is linearly identifiable, it is also affinely identifiable; the other direction is not necessarily true.

Both types of identifiability are important properties in the design of machine-learning solutions [24]. Particularly, in the case of the hypothesis class Lin and the cost function *lsq* (as defined in Section 2), linear independence implies that there is a single optimal hypothesis, whereas its absence implies that the space of optimal solutions is unbounded. Affine independence likewise arises in different cost functions such as *maximum entropy* [32]. The corresponding computational problem is formally defined as follows.

PROBLEM 3 (IDENTIFIABILITY). *Let* **QL** *be a query language. The computational problem of* linear *(respectively,* affine*)* **QL**-identifiability *is that of testing, given an entity schema* $\mathbf{S}$ *and a statistic* $\Pi$ *over* $\mathbf{S}$*, whether* $\Pi$ *is linearly (respectively, affinely) identifiable.*

EXAMPLE 4.3. Consider again $\mathbf{S}$ and $\Pi$ of our running example (Figure 1). Then $\mathbf{S}$ and $\Pi$ form a "yes" instance of linear (and affine) CQ-identifiability. Indeed, $\Pi$ is linearly (and affinely) identifiable, and a witness instance is $I$ of Figure 2 with Txn restricted to the entities 1 and 2 (or 3 and 2, but not 1 and 3). We have shown that under certain conditions (that hold in our case), a statistic that consists of CQ feature queries is always identifiable, unless two or more of the feature queries are equivalent [22]. Hence, in the case of CQs, identifiability "comes for free" up to redundancy. □

## 4.4 Complexity Analysis

Complexity analysis of the three problems can be found in the conference version of this article [22], and will be presented in more detail in the full version of the paper. In the next section, we give complexity results on the first problem, namely separability.

# 5. COMPLEXITY OF SEPARABILITY

In this section, we discuss the complexity of separability in the case where feature queries are from the class of CQs and the hypothesis class is that of linear classifiers. The first result states coNP-completeness.

THEOREM 5.1. $(\mathsf{Lin}, \mathsf{CQ})$*-separability is a* coNP*-complete problem. Moreover, there exists a fixed entity schema* $\mathbf{S}$ *such that* $(\mathsf{Lin}, \mathsf{CQ})$*-separability is coNP-hard over* $\mathbf{S}$*.*

The proof of Theorem 5.1 consists of two parts. In the first part, we show that a given $(I, \lambda)$ is Lin-separable w.r.t. a given statistic $\Pi$ if and only if every two entities with different labels (i.e., one is $+1$ and the other $-1$) can be *distinguished* by a CQ, that is, there is a CQ feature query that returns one entity and not the other. The second part shows that this distinguishability test is coNP-complete. This proof highlights a connection to the problems of *query-by-example* and *definability* [7, 30, 33], and we are currently exploring these connections in more depth.

In the above proof of hardness we construct CQs over a fixed schema, but self joins are allowed. Next, we consider the case of self-join-free CQs. Formally, a CQ $q$ is *self-join free* if it does not have two distinct atoms with the same relation symbol. We denote by $\mathsf{CQ}_{\mathsf{sjf}}$ the class of CQs without self joins. Interestingly, disallowing self joins (hence, restricting the space of statistics to simpler CQs) does not make the problem easier. In fact, under conventional complexity assumptions, it becomes harder!

THEOREM 5.2. $(\mathsf{Lin}, \mathbf{QL}_{\mathsf{sjf}})$*-separability is* $\Sigma_2^{\mathrm{P}}$*-complete.*

Intuitively, the reason for the increased complexity is that self joins allow us to (efficiently) formulate a single statistic $\Pi$ of representative ("canonical") feature CQs that captures the entire space of statistics; that is, if any statistic provides separation, then so does $\Pi$. In particular, with self joins the problem boils down to deciding on the existence of a homomorphism. Yet, without self joins it appears that we cannot do better than to inspect an exponential space of statistics, and solve the homomorphism problem in each. Finally, we remark that fixing the schema $\mathbf{S}$ in the case of $\mathbf{QL}_{\mathsf{sjf}}$ would make the separability problem solvable in polynomial time, since the number of possible statistics (without equivalent feature queries) is bounded by a fixed constant, and each feature query can be evaluated in polynomial time.

COMMENT 5.3. For CQs with constants, separability is trivial, since the positive examples can be hardcoded into the statistic. In Figure 2, for instance, we could encode each of the first, second, and third tuples of TxnInfo (and even their join with Card) in a CQ that selects precisely the corresponding transactions. It would, however, be interesting to enforce restrictions on the usage of constants (e.g., limit their number). An elegant way to formalize such restrictions was taken by Grohe and Ritzert [16] that separate the variables into *ordinary variables* and *parameters* that can be set fixed by the learning algorithm. We plan to explore this approach in future work. □

# 6. ADDITIONAL RELATED WORK

The task of feature engineering has been widely studied for decades [9, 17–20]. Our approach borrows heavily from the feature-engineering process identified in Guyon's seminal book [18] and those we have observed in practice. Feature engineering has received some attention from the database community [2, 3, 23, 29, 35]. That work has made algorithmic or tooling contributions to better support feature engineering, while it has not addressed the fundamental questions that our framework targets.

Frameworks and query languages that fuse logic with probabilistic semantics, to simplify the design of machine-learning models, have been proposed and developed in past decades. Examples of these include Probabilistic Relation Models pioneered by Koller and Friedman [14], PRISM [27], BLOG [25], Markov Logic Networks [26], and the recent *Probabilistic-Programming Datalog* [6]. However, these approaches focus on orthogonal formal questions: the semantics of the models and the complexity of the associated inference tasks. In contrast, we consider the interplay of the logical rules and learning properties. In particular, to the best of our knowledge this work is the first to consider separability, identifiability, and dimensionality in machine-learning models that are defined over database queries. Our formal framework draws inspiration from previous approaches to combining logical reasoning to probabilistic reasoning, which is a classic topic [5, 11], but is distinct in its goal.

There has been a lot of work in the Machine Learning community on learnability aspects of First Order formulas. For

instance, Arias and Khardon [4] considered such aspects (including VC dimension) in the context of Horn clauses, where they establish bounds that are based on syntactic properties of the clauses (e.g., number of variables, literals, clauses, etc.). Similarly, Grohe and Ritzert [16] explored PAC learning of first-order formulas over a "background structure," namely a database. Such setups are quite different from ours, since their goal is to classify a whole interpretation (database) based on a single formula (to be learned), while we consider classification of entities within a single database and focus on *feature engineering* rather than the engineering of the classifier. More technically, in our framework the goal is not necessarily to learn queries, but rather to reason about queries as features of machine-learning models that are not necessarily database queries (e.g., linear models).

# 7. CONCLUSIONS AND DIRECTIONS

We described a framework for feature engineering towards programming machine-learning solutions over a database, while focusing on the important task of classification. Our framework is based on simple additions to the relational data and query model, where an *entity schema* allows to represent entities along with their associated information, and where feature engineering is the task of designing a *statistic* when given a training instance over the entity schema. This framework enables us to formalize relevant computational problems, conduct nontrivial analyses, and reach insights and solutions. In particular, we have formalized three important computational problems within the framework: separability, identifiability, and VC dimensionality. These problems are parameterized by the hypothesis class in use and the query language deployed for feature extraction.

Focusing on features definable as conjunctive queries and on linear hypotheses, we have drawn connections between the studied computational problems and those of query containment and equivalence. These connections have several interesting consequences. For one, there is a tight relationship between the computational complexity of our problems and that of query containment: it is necessary, and often sufficient, to solve CQ containment in order to solve our problems. Moreover, the fact that identifiability "comes for free" (up to redundancy) gives a formal indication of the suitability of CQs as a language for feature engineering. It also motivates the challenge of finding other natural query languages that are likewise suitable. We conclude with a number of directions and extensions for future research.

## *Logical Analysis*

**Further expressiveness.** We have focused on the simple class of conjunctive queries for defining statistics, and on the classifier class of linear hypotheses. An immediate future direction would be to consider more expressive classes. For features, these can be unions of conjunctive queries, queries with additional logical operators, non-monotonic features, and aggregate functions. For the hypotheses, future directions can consider any standard class, such as decision trees.

**Schema constraints.** The complexity of some of the tasks we have considered would be impacted by the presence of schema constraints. In particular, in the identifiability problem column independence would need to be realized by an instance that satisfies the constraints, and not by any instance of the signature. The problem of VC dimensionality would be similarly impacted. We view this direction as an important opportunity of incorporating the database's rich modeling of data into the task of feature engineering.

**Text analysis.** An area where machine-learning classification is crucial for even simple tasks is that of text analysis, and in particular when the text is in natural language from open domains such as Web and social media [29]. Consequently, we belive that a direction of a high potential impact is that of applying our framework to formalisms that involve queries over text, such as the *document spanners* of Fagin et al. [12, 13] that construct and manipulate relations over text spans (intervals) using extractors (e.g., regular expressions). In particular, the computational challenges will involve queries with both relational and textual operations.

## *Statistical Questions*

**Generalized learning tasks.** Our features in this work were all Boolean ($\pm 1$), and it is desirable to study the natural extension of the framework to numerical features, where numbers are either directly copied from the database or indirectly computed via queries. Numerical values will likely complicate the basic model, as they entail arguing about schema constraints to ensure that a feature query associates a unique value with each entity. Orthogonally, the framework can be generalized to other prediction tasks, such as multi-label classification (e.g., predict the age group of a person) and numerical regression (e.g., predict the actual age of the person). It is important to understand how the challenges we considered are affected by such generalizations.

**Separability relaxation.** The separability problem, as defined in this article, can be extended by allowing for an *approximate* agreement with the training examples (e.g., the hypothesis $h$ should agree with the labeling $\lambda$ on at least $(1-\epsilon)$ of the entities, or at most $k$ entities should be misclassified). This is a practical and crucial relaxation in practical scenarios. For one, the training data may be noisy. Moreover, our hypothesis class may be too simple to precisely cover the examples, but can do so with only a small error.

**Model complexity.** While extending the expressiveness of queries, it is of high importance to find the proper *restrictions* on the engineered statistics (a.k.a. *regularization*), in order to *(a)* reduce the model complexity and, consequently, reduce the risk of overfitting to the training samples, *and (b)* gain more efficient machine-leaning solutions. The common regularization limits the length of the statistic; in our framework, we can consider restrictions on the feature queries, such as size, structure, number of constants/variables, and so on. The ultimate goal is to find settings that properly balance between overfitting, underfitting, inference (classification) complexity and learning (training) complexity.

The vast literature on machine learning gives rise to many more directions for our framework to extend, such as notions of capacity beyond VC dimension (e.g., *Rademacher* and *Gaussian* complexities [8]) and the implications of the "transductive" learning environments, where we know to begin with what entities we will need to predict upon [15]. We believe that our framework can contribute to many of these directions the important angle of data and query modelling.

## Acknowledgments

The authors are grateful to Stephen Bach and Alex Ratner for insightful input on this work, and for Jared Alexander Dunnmon for valuable suggestions on this article.

## 8. REFERENCES

[1] *SAS Report on Analytics*. `sas.com/reg/wp/corp/23876`.

[2] M. Anderson, D. Antenucci, V. Bittorf, M. Burgess, M. Cafarella, A. Kumar, F. Niu, Y. Park, C. Ré, and C. Zhang. Brainwash: A Data System for Feature Engineering. In *CIDR*, 2013.

[3] M. R. Anderson, M. J. Cafarella, Y. Jiang, G. Wang, and B. Zhang. An integrated development environment for faster feature engineering. *PVLDB*, 7(13):1657–1660, 2014.

[4] M. Arias and R. Khardon. Complexity parameters for first order classes. *Machine Learning*, 64(1-3):121–144, 2006.

[5] F. Bacchus, A. J. Grove, J. Y. Halpern, and D. Koller. From statistical knowledge bases to degrees of belief. *CoRR*, cs.AI/0307056, 2003.

[6] V. Bárány, B. ten Cate, B. Kimelfeld, D. Olteanu, and Z. Vagena. Declarative probabilistic programming with datalog. In *ICDT*, volume 48 of *LIPIcs*, pages 7:1–7:19. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.

[7] P. Barceló and M. Romero. The complexity of reverse engineering problems for conjunctive queries. In *ICDT*, volume 68 of *LIPIcs*, pages 7:1–7:17. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017.

[8] P. L. Bartlett and S. Mendelson. Rademacher and gaussian complexities: Risk bounds and structural results. In *COLT*, pages 224–240, 2001.

[9] D. E. Boyce. *Optimal Subset Selection: Multiple Regression, Interdependence, and Optimal Network Algorithms* . Springer-Verlag, 1974.

[10] C. Dwork, M. Hardt, T. Pitassi, O. Reingold, and R. S. Zemel. Fairness through awareness. In *ITCS*, pages 214–226. ACM, 2012.

[11] R. Fagin, J. Y. Halpern, and N. Megiddo. A logic for reasoning about probabilities. *Inf. Comput.*, 87(1/2):78–128, 1990.

[12] R. Fagin, B. Kimelfeld, F. Reiss, and S. Vansummeren. Spanners: a formal framework for information extraction. In *PODS*, pages 37–48, 2013.

[13] R. Fagin, B. Kimelfeld, F. Reiss, and S. Vansummeren. Document spanners: A formal approach to information extraction. *J. ACM*, 62(2):12, 2015.

[14] N. Friedman, L. Getoor, D. Koller, and A. Pfeffer. Learning probabilistic relational models. In *IJCAI*, pages 1300–1309, 1999.

[15] A. Gammerman, K. S. Azoury, and V. Vapnik. Learning by transduction. In *UAI*, pages 148–155. Morgan Kaufmann, 1998.

[16] M. Grohe and M. Ritzert. Learning first-order definable concepts over structures of small degree. In *LICS*, pages 1–12. IEEE Computer Society, 2017.

[17] I. Guyon and A. Elisseeff. An introduction to variable and feature selection. *Journal of Machine Learning Research*, 3:1157–1182, 2003.

[18] I. Guyon, S. Gunn, M. Nikravesh, and L. A. Zadeh. *Feature Extraction: Foundations and Applications (Studies in Fuzziness and Soft Computing)*. Springer-Verlag New York, Inc., 2006.

[19] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning: Data mining, inference, and prediction*. Springer, 2001.

[20] G. H. John, R. Kohavi, and K. Pfleger. Irrelevant features and the subset selection problem. In *Machine Learning*, pages 121–129, 1994.

[21] S. Kandel, A. Paepcke, J. M. Hellerstein, and J. Heer. Enterprise data analysis and visualization: An interview study. *IEEE Trans. Vis. Comput. Graph.*, 18(12):2917–2926, 2012.

[22] B. Kimelfeld and C. Ré. A relational framework for classifier engineering. In *PODS*, pages 5–20. ACM, 2017.

[23] A. Kumar, J. F. Naughton, J. M. Patel, and X. Zhu. To join or not to join?: Thinking twice about joins before feature selection. In *SIGMOD Conference*, pages 19–34. ACM, 2016.

[24] E. L. Lehmann and G. Casella. *Theory of point estimation*, volume 31. Springer, 1998.

[25] B. Milch, B. Marthi, S. J. Russell, D. Sontag, D. L. Ong, and A. Kolobov. Blog: Probabilistic models with unknown objects. In *IJCAI*, pages 1352–1359, 2005.

[26] M. Richardson and P. Domingos. Markov logic networks. *Mach. Learn.*, 62(1-2):107–136, 2006.

[27] T. Sato and Y. Kameya. PRISM: A language for symbolic-statistical modeling. In *IJCAI*, pages 1330–1339, 1997.

[28] S. Shalev-Shwartz and S. Ben-David. *Understanding Machine Learning: From Theory to Algorithms*. Cambridge University Press, 2014.

[29] J. Shin, S. Wu, F. Wang, C. D. Sa, C. Zhang, and C. Ré. Incremental knowledge base construction using DeepDive. *PVLDB*, 8(11):1310–1321, 2015.

[30] B. ten Cate and V. Dalmau. The product homomorphism problem and applications. In *ICDT*, volume 31 of *LIPIcs*, pages 161–176. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.

[31] V. N. Vapnik and A. Y. Chervonenkis. On the uniform convergence of relative frequencies of events to their probabilities. *Theory of Probability and its Applications*, 16(2):264–280, 1971.

[32] M. J. Wainwright and M. I. Jordan. Graphical models, exponential families, and variational inference. *Foundations and Trends in Machine Learning*, 1(1-2):1–305, 2008.

[33] R. Willard. Testing expressibility is hard. In *CP*, volume 6308 of *Lecture Notes in Computer Science*, pages 9–23. Springer, 2010.

[34] R. S. Zemel, Y. Wu, K. Swersky, T. Pitassi, and C. Dwork. Learning fair representations. In *ICML*, volume 28 of *JMLR Proceedings*, pages 325–333. JMLR.org, 2013.

[35] C. Zhang, A. Kumar, and C. Ré. Materialization optimizations for feature selection workloads. In *SIGMOD Conference*, pages 265–276, 2014.

[36] I. Zliobaite. A survey on measuring indirect discrimination in machine learning. *CoRR*, abs/1511.00148, 2015.

# Technical Perspective: From Think Parallel to Think Sequential

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

In recent years, the database and distributed systems communities have built a wide variety of runtime systems and programming models for large-scale computing over graphs. Such "big graph processing systems" [1, 2, 4, 5, 7] o support highly scalable parallel execution of graph algorithms — e.g., computing shortest paths, graph centrality, connected components, or perhaps even graph clusters. As described in the excellent survey by Yan et al [6], most big graph processing systems require the programmer to adopt a *vertex-centric* or *block-centric* programming model. For the former, code only "sees" the state at one vertex, receives messages from other vertices, and can send messages to other vertices. Under the latter, code manages a set of vertices within a subgraph ("block") and can communicate with the code managing other blocks.

In "From think Parallel to Think Sequential," Fan and colleagues argue that vertex- and block-centric programming models are not natural for programmers trained to think sequentially. Instead, they argue that a more intuitive programming model can be developed out of several very simple primitives that can be composed to do incremental computation (as has also been studied in more general "big data" systems [4, 3]). The authors propose four elegant building blocks: (1) a *partial evaluation* function, (2) an *incremental update* handling function, (3) mechanisms for updating and sharing parameters in global fashion, and (4) an *aggregate function* for when multiple workers are updating the same parameter. They build the GRAPE GRAPh Engine system, which implements this programming model, and they show that it provides excellent performance for a variety of graph algorithms.

The paper presents a compelling case that, at least for certain classes of algorithms, the simple primitives may be both more natural and more amenable to optimization than standard vertex-centric approaches.

## 1. REFERENCES

[1] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, and Carlos Guestrin. Graphlab: A distributed framework for machine learning in the cloud. *CoRR*, abs/1107.0922, 2011.

[2] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, pages 135–146, 2010.

[3] Svilen Mihaylov, Zachary G. Ives, and Sudipto Guha. REX: Recursive, delta-based data-centric computation. In *PVLDB*, 2012.

[4] Derek Gordon Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: a timely dataflow system. In *SOSP*, pages 439–455, 2013.

[5] Reynold S Xin, Joseph E Gonzalez, Michael J Franklin, and Ion Stoica. Graphx: A resilient distributed graph system on spark. In *First International Workshop on Graph Data Management Experiences and Systems*, page 2. ACM, 2013.

[6] Da Yan, Yingyi Bu, Yuanyuan Tian, and Amol Deshpande. Big graph analytics platforms. *Foundations and Trends® in Databases*, 7(1-2):1–195, 2017.

[7] Da Yan, James Cheng, Yi Lu, and Wilfred Ng. Blogel: A block-centric framework for distributed computation on real-world graphs. *Proceedings of the VLDB Endowment*, 7(14):1981–1992, 2014.

# From Think Parallel to Think Sequential

Wenfei Fan[1,2], Yang Cao[1], Jingbo Xu[2], Wenyuan Yu[2], Yinghui Wu[3],
Chao Tian[1,2], Jiaxin Jiang[4], Bohan Zhang[5]
[1]Univ. of Edinburgh [2]Beihang Univ. [3]Washington State Univ. [4]Hong Kong Baptist Univ. [5]Peking Univ.
{wenfei@inf, yang.cao@, chao.tian@}ed.ac.uk, {xujb, yuwenyuan}@act.buaa.edu.cn, yinghui@eecs.wsu.edu,
jxjian@comp.hkbu.edu.hk, bohan@pku.edu.cn

## ABSTRACT

This paper presents GRAPE, a parallel GRAPh Engine for graph computations. GRAPE differs from previous graph systems in its ability to parallelize existing sequential graph algorithms as a whole, without the need for recasting the entire algorithms into a new model. Underlying GRAPE are a simple programming model, and a principled approach based on fixpoint computation with partial evaluation and incremental computation. Under a monotonic condition, GRAPE guarantees to converge at correct answers as long as the sequential algorithms are correct. We show how our familiar sequential graph algorithms can be parallelized by GRAPE. In addition to the ease of programming, we experimentally verify that GRAPE achieves comparable performance to the state-of-the-art graph systems, using real-life and synthetic graphs.

## 1. INTRODUCTION

There has been increasing demand for graph computations, *e.g.,* graph traversal, connectivity, pattern matching, and collaborative filtering. Indeed, graph computations have found prevalent use in mobile network analysis, pattern recognition, knowledge discovery, transportation networks, social media marketing and fraud detection, among other things. In addition, real-life graphs are typically big, easily having billions of nodes and trillions of edges [18]. With these comes the need for parallel graph computations. In response to the need, several parallel graph systems have been developed, *e.g.,* Pregel [25], GraphLab [16, 24], Trinity [29], GRACE [35], Blogel [37], Giraph++ [31], and GraphX [17].

However, users often find it hard to write and debug parallel graph programs using these systems. The most popular programming model for parallel graph algorithms is the vertex-centric model, pioneered by Pregel and GraphLab. For instance, to program with Pregel, one needs to "think like a vertex", by writing a user-defined function *compute(msgs)* to be executed at a vertex $v$, where $v$ communicates with other vertices by message passing (*msgs*). Although graph computations have been studied for decades and a large number of sequential (single-machine) graph algorithms are already in place, to use Pregel, one has to recast the existing algorithms into vertex-centric programs. Trinity and GRACE also support vertex-centric programming. While Blogel and Giraph++ allow blocks to have their status as a "vertex" and support block-level communication, they still adopt the vertex-centric programming paradigm. GraphX also recasts graph computation into its distributed dataflow framework as a sequence of join and group-by stages punctuated by map operations, on Spark platform (see [36] for a survey). The recasting is nontrivial for users who are not very familiar with the parallel models. Moreover, none of the systems provides guarantee on the correctness or even termination of parallel programs developed in their models. These make the existing systems a privilege for experienced users only.



| System | Category | Time(s) | Comm.(MB) |
|---|---|---|---|
| Giraph | vertex-centric | 434.0 | $1.13 \times 10^5$ |
| GraphLab | vertex-centric | 41.7 | $1.07 \times 10^5$ |
| Blogel | block-centric | 112.3 | $1.23 \times 10^5$ |
| GRAPE | think sequential | 24.3 | $1.47 \times 10^4$ |

Table 1: Graph traversal on parallel systems

Is it possible to simplify parallel programming for graph computations, from "think parallel" to "think sequential"? That is, can we have a system that parallelizes existing sequential graph algorithms across a cluster of processors? Better yet, is there a general condition under which the parallelization guarantees to converge at correct answers as long as the sequential algorithms are correct? After all, the human's brain is not wired to think parallel.

To answer these questions, we develop GRAPE, a parallel GRAPh Engine. It differs from prior systems in the following.

*(1) Ease of programming*. GRAPE supports a simple programming model. For a class $\mathcal{Q}$ of graph queries, users only need to provide three sequential (incremental) algorithms for $\mathcal{Q}$, with *no need* to recast them into a new model, or revise the logic of the algorithms. This makes parallel computations accessible to users who know conventional graph algorithms covered in college textbooks.

*(2) Parallelization*. GRAPE *parallelizes* the computation across a cluster of processors, based on a fixpoint computation with partial evaluation and incremental computation. Under a monotonic condition, it guarantees to converge with correct answers as long as the three sequential algorithms provided are correct.

*(3) Optimization*. GRAPE inherits all optimization strategies available for sequential graph algorithms, *e.g.,* indexing, compression and partitioning. These are hard to implement for vertex programs.

*(4) Scale-up*. The ease of programming does not imply performance degradation compared with the state-of-the-art systems such as vertex-centric Giraph [3] (Pregel) and GraphLab, and block-centric Blogel. For instance, Table 1 shows the performance of the systems for shortest-path queries over Friendster [2] with 192 workers. GRAPE outperforms Giraph, GraphLab and Blogel in both response time and communication costs (see Section 4).

This paper presents the programming and parallel models of GRAPE (Section 2), shows how it parallelizes sequential algorithms (Section 3), and empirically evaluates GRAPE (Section 4).

# 2. GRAPE PARALLELIZATION

We present the programming paradigm and parallel model of GRAPE. Interested readers are invited to see [14] for details.

## 2.1 Graph Partition

We start with basic notations. We consider directed or undirected graphs $G = (V, E, L)$, where (1) $V$ is a finite set of nodes; (2) $E \subseteq V \times V$ is a set of edges; and (3) each node $v$ in $V$ (resp. edge $e \in E$) carries $L(v)$ (resp. $L(e)$), indicating its content, as found in social networks, knowledge bases and property graphs.

**Partition strategy**. Given a graph $G$ and an integer $m$, a graph partition strategy $\mathcal{P}$ partitions $G$ into *fragments* $\mathcal{F} = (F_1, \ldots, F_m)$. Each fragment $F_i = (V_i, E_i, L_i)$ is a subgraph of $G$ that resides at processor $P_i$, for $i \in [1, m]$; and $E = \bigcup_{i\in[1,m]} E_i$, $V = \bigcup_{i\in[1,m]} V_i$. Under edge-cut partition [8, 9], denote by

- $F_i.I$ the set of nodes $v \in V_i$ such that there exists edge $(v', v)$ *from* a node $v'$ in $F_j$;
- $F_i.O$ the set of nodes $v'$ in some $F_j$ such that there exists an edge $(v, v')$ from $v \in V_i$; and
- $\mathcal{F}.O = \bigcup_{i\in[1,m]} F_i.O$, and $\mathcal{F}.I = \bigcup_{i\in[1,m]} F_i.I$.

A cut edge from $F_i$ to $F_j$ has a copy in each of $F_i$ and $F_j$ ($i \neq j$). We refer to nodes in $F_i.I$ (or $F_i.O$) as *border nodes* of fragment $F_i$ *w.r.t.* partition strategy $\mathcal{P}$. Note that $\mathcal{F}.I = \mathcal{F}.O$.

Under vertex-cut partition [22], $\mathcal{F}.O$ and $\mathcal{F}.I$ correspond to entry vertices and exit vertices, respectively.

## 2.2 Programming Paradigm

Consider a graph computation problem $\mathcal{Q}$. Using our familiar terms, we refer to an instance $Q$ of $\mathcal{Q}$ as a *query* of $\mathcal{Q}$. To answer $Q \in \mathcal{Q}$ with GRAPE, a user only needs to specify three functions.

PEval: a sequential algorithm for $\mathcal{Q}$ that given a query $Q \in \mathcal{Q}$ and a graph $G$, computes the answer $Q(G)$ to $Q$ in $G$.

IncEval: a sequential algorithm IncEval for $\mathcal{Q}$ that given $Q$, $G$, $Q(G)$ and updates $\Delta G$ to $G$, incrementally computes changes $\Delta O$ to the old output $Q(G)$ such that $Q(G \oplus \Delta G) = Q(G) \oplus \Delta O$, where $G \oplus \Delta G$ denotes graph $G$ updated by $\Delta G$.

Assemble: a function Assemble that collects partial answers computed locally at each worker by PEval and IncEval, and assembles them into complete answer $Q(G)$. It is typically straightforward.

Functions PEval, IncEval and Assemble are referred to as *a* PIE *program for* $\mathcal{Q}$. Here PEval and IncEval are *existing sequential* (incremental) algorithms for $\mathcal{Q}$, with the following additions to PEval.

*Update parameters*. PEval declares *status variables* $\bar{x}$ for a set $C_i$ of nodes and edges in a fragment $F_i$, which store contents of $F_i$ or intermediate results of a computation. Here $C_i$ is a set of nodes and edges within $d$-hops of the border nodes in $F_i$, *e.g.,* $F_i.O$, for an integer $d$. When $d = 0$, one may define $C_i$ as, *e.g.,* $F_i.O$.

We denote by $C_i.\bar{x}$ the set of *update parameters* of $F_i$, which consists of status variables of the nodes and edges in $C_i$, *i.e.,* variables in $C_i.\bar{x}$ are the candidates to be updated.

*Aggregate function*. PEval also specifies a function $f_{\mathsf{aggr}}$, *e.g.,* min, max, to resolve conflicts when multiple workers attempt to assign different values to the same update parameter.

The update parameters and aggregate function are specified in PEval and are shared by IncEval. As will be seen shortly, IncEval only needs to deal with changes $\Delta G$ to update parameters.

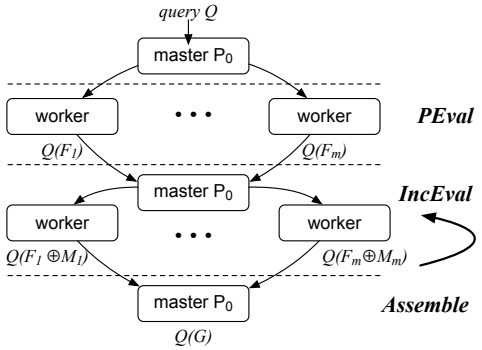



Figure 1: Workflow of GRAPE

## 2.3 Parallel Model

Given a partition strategy $\mathcal{P}$ and a PIE program $\rho$ (PEval, IncEval, Assemble) for $\mathcal{Q}$, GRAPE parallelizes $\rho$ as follows. It first partitions $G$ into $(F_1, \ldots, F_m)$ with $\mathcal{P}$, and distributes fragments $F_i$'s across $m$ shared-nothing *virtual workers* $(P_1, \ldots, P_m)$. It maps $m$ virtual workers to $n$ physical workers. When $n < m$, multiple virtual workers mapped to the same worker share memory. Graph $G$ is partitioned *once* for *all queries* $Q \in \mathcal{Q}$ on $G$.

We start with basic ideas behind GRAPE parallelization.

*Partial evaluation*. Given a function $f(s, d)$ and the $s$ part of its input, *partial evaluation* is to specialize $f(s, d)$ *w.r.t.* the known input $s$ [21]. That is, it performs the part of $f$'s computation that depends only on $s$, and generates a partial answer, *i.e.,* a residual function $f'$ that depends on the as yet unavailable input $d$. For each worker $P_i$ in GRAPE, its local fragment $F_i$ is its known input $s$, while the data residing at other workers accounts for the yet unavailable input $d$. As will be seen shortly, given a query $Q \in \mathcal{Q}$, GRAPE computes $Q(F_i)$ in parallel as partial evaluation.

*Incremental evaluation*. Workers exchange *changed values* of their local update parameters with each other. Upon receiving message $M_i$ that consists of changes to the update parameters at fragment $F_i$, worker $P_i$ treats $M_i$ as *updates* to $F_i$, and *incrementally* computes changes $\Delta O_i$ to $Q(F_i)$ such that $Q(F_i \oplus M_i) = Q(F_i) \oplus \Delta O_i$. This is often more efficient than recomputing $Q(F_i \oplus M_i)$ starting from scratch, since in practice $M_i$ is often small. Better still, the computation may be *bounded*: its cost can be expressed as a function in $|M_i| + |\Delta O_i|$, *i.e.,* the size of changes in the input and output, instead of $|F_i|$, no matter how big $F_i$ is [12, 28].

**Parallelization**. We use (BSP) (Bulk Synchronous Parallel model [32]). Given a query $Q \in \mathcal{Q}$ at master $P_0$, GRAPE answers $Q$ in the partitioned graph $G$. It posts the same $Q$ to all the workers, and computes $Q(G)$ in three phases as follows, as shown in Fig. 1.

*(1) Partial evaluation* (**PEval**). In the first superstep, upon receiving query $Q$, each worker $P_i$ applies function PEval to its local fragment $F_i$, to compute partial results $Q(F_i)$, in parallel ($i \in [1, m]$). After $Q(F_i)$ is computed, PEval generates a message at each worker $P_i$ and sends it to master $P_0$. The message is simply the set $C_i.\bar{x}$ of update parameters at fragment $F_i$.

For each $i \in [1, m]$, master $P_0$ maintains update parameters $C_i.\bar{x}$. It deduces a message $M_i$ to worker $P_i$ based on the following *message grouping policy*. (a) For each status variable $x \in C_i.\bar{x}$, it collects the group $S_x$ of values for $x$ from all messages, and computes $x_{\mathsf{aggr}} = f_{\mathsf{aggr}}(S_x)$ by applying the aggregate function $f_{\mathsf{aggr}}$. (b) Message $M_i$ includes only those $x_{\mathsf{aggr}}$'s such that $x_{\mathsf{aggr}} \neq x$, *i.e.,* only those *changed* values of the update parameters at $F_i$.

*(2) Incremental computation* (**IncEval**). GRAPE iterates the following supersteps until it terminates. Following BSP, each super-

*Input:* $F_i(V_i, E_i, L_i)$, source vertex $s$
*Output:* $Q(F_i)$ consisting of current $\mathsf{dist}(s, v)$ for all $v \in V_i$

*Declaration*: /*candidate set $C_i$ is $F_i.O$*/
**for** each node $v \in V_i$, an integer variable $\mathsf{dist}(s, v)$;
message $M_i := \{\mathsf{dist}(s, v) \mid v \in F_i.O\}$;
aggregate function $f_{\mathsf{aggr}} = \mathsf{min}(\mathsf{dist}(s, v))$;

/*sequential algorithm for SSSP (pseudo-code)*/
1. initialize priority queue Que;
2. $\mathsf{dist}(s, s) := 0$;
3. **for each** $v$ in $V_i$ **do**
4. **if** $v! = s$ **then**
5. $\mathsf{dist}(s, v) := \infty$;
6. Que.addOrAdjust($s$, $\mathsf{dist}(s, s)$);
7. **while** Que is not empty **do**
8. $u$ := Que.pop() // pop vertex with minimal distance
9. **for each** child $v$ of $u$ **do** // only $v$ that is still in Que
10. $alt := \mathsf{dist}(s, u) + L_i(u, v)$;
11. **if** $alt < \mathsf{dist}(s, v)$ **then**
12. $\mathsf{dist}(s, v) := alt$;
13. Que.addOrAdjust($v$, $\mathsf{dist}(s, v)$);
14. $Q(F_i) := \{\mathsf{dist}(s, v) \mid v \in V_i\}$

Figure 2: Parallel SSSP: Partial evaluation PEval

step starts after the master $P_0$ receives messages (possibly empty) from *all workers* $P_i$ for $i \in [1, m]$. A superstep has two steps itself, one at $P_0$ and the other at the workers.

(a) Master $P_0$ routes (nonempty) messages from the last superstep to workers, if there exists any.

(b) Upon receiving message $M_i$, worker $P_i$ *incrementally* computes $Q(F_i \oplus M_i)$ by applying IncEval, and by *treating* $M_i$ *as updates* to $C_i.\bar{x}$, in parallel for $i \in [1, m]$.

At the end of the process of IncEval, worker $P_i$ sends a message to $P_0$ that encodes *updated values* of $C_i.\bar{x}$, if any. Upon receiving messages from all workers, master $P_0$ deduces message $M_i$ to each worker $P_i$ following the message grouping policy given above; it sends message $M_i$ to worker $P_i$ in the next superstep.

*(3) Termination* (**Assemble**). At each superstep, master $P_0$ checks whether for all $i \in [1, m]$, $P_i$ is inactive, *i.e.,* $P_i$ is done with its local computation, and there exists no more change to the update parameters of $F_i$. If so, GRAPE pulls partial results from all workers, and applies Assemble to group them together and get the final result at $P_0$, denoted by $\rho(Q, G)$. It returns $\rho(Q, G)$ and terminates.

**Example 1:** We show how GRAPE parallelizes the computation of Single Source Shortest Path (SSSP), a common graph computation problem. Consider a directed graph $G = (V, E, L)$ in which for each edge $e$, $L(e)$ is a positive number. The length of a path $(v_0, \ldots, v_k)$ in $G$ is the sum of $L(v_{i-1}, v_i)$ for $i \in [1, k]$. For a pair $(s, v)$ of nodes, denote by $\mathsf{dist}(s, v)$ the *distance* from $s$ to $v$, *i.e.,* the length of a shortest path from $s$ to $v$. Given graph $G$ and a node $s$ in $V$, SSSP computes $\mathsf{dist}(s, v)$ for all $v \in V$.

Under edge-cut partition [9], GRAPE takes the set $F_i.O$ of "border nodes" as $C_i$ at each $P_i$ (with edges across distinct fragments). The PIE program for SSSP consists of (1) our familiar Dijkstra's algorithm for SSSP [15] as PEval, (2) a sequential incremental algorithm of [27] as IncEval, and (3) a straightforward Assemble.

*(1) PEval*. As shown in Fig. 2, PEval (lines 1-14) is verbally identical to Dijsktra's algorithm [15]. One only needs to declare (a) status variable as an integer variable $\mathsf{dist}(s, v)$ for each node $v$, initially $\infty$ (except $\mathsf{dist}(s, s) = 0$); (b) update parameters as $C_i.\bar{x} = \{\mathsf{dist}(s, v) \mid v \in F_i.O\}$, *i.e.,* the status variables associated with nodes in $F_i.O$ at fragment $F_i$; and (c) min as an aggregate function $f_{\mathsf{aggr}}$. If there are multiple values for the same $\mathsf{dist}(s, v)$, the smallest value is taken by the order on positive numbers.

At the end of its process, PEval sends $C_i.\bar{x}$ to master $P_0$. At $P_0$, GRAPE maintains $\mathsf{dist}(s, v)$ for all $v \in F.O = F.I$. Upon receiving messages from all workers, it takes the smallest value for

*Input:* $F_i(V_i, E_i, L_i)$, partial result $Q(F_i)$, message $M_i$
*Output:* $Q(F_i \oplus M_i)$

*Declaration*: message $M_i = \{\mathsf{dist}(s, v) \mid v \in F_i.O,\ \mathsf{dist}(s, v)$ decreased$\}$;

1. initialize priority queue Que;
2. **for each** $\mathsf{dist}(s, v)$ in $M_i$ **do**
3. Que.addOrAdjust($v$, $\mathsf{dist}(s, v)$);
4. **while** Que is not empty **do**
5. $u$ := Que.pop() /* pop vertex with minimum distance*/
6. **for each** children $v$ of $u$ **do**
7. $alt := \mathsf{dist}(s, u) + L_i(u, v)$;
8. **if** $alt < \mathsf{dist}(s, v)$ **then**
9. $\mathsf{dist}(s, v) := alt$;
10. Que.addOrAdjust($v$, $\mathsf{dist}(s, v)$);
11. $Q(F_i) := \{\mathsf{dist}(s, v) \mid v \in V_i\}$

Figure 3: Parallel SSSP: Incremental evaluation IncEval

each $\mathsf{dist}(s, v)$. It finds those variables with smaller $\mathsf{dist}(s, v)$ for $v \in F_j.O$, groups them into message $M_j$, and sends $M_j$ to $P_j$.

*(2) IncEval*. We give IncEval in Fig. 3. It is the sequential incremental algorithm for SSSP in [28] that is mildly revised to deal with changed $\mathsf{dist}(s, v)$ for $v$ in $F_i.I$ (deduced by leveraging $\mathcal{F}.I = \mathcal{F}.O$). Using a queue Que, it starts with changes in $M_i$, propagates the changes to affected area, and updates the distances (see [28]). The partial result now consists of the revised distances (line 11). At the end of the process, it sends to master $P_0$ the updated values of those status variables in $C_i.\bar{x}$, as in PEval. It applies the aggregate function min to resolve conflicts.

Following [28], one can show that IncEval is *bounded*: its cost is determined by the sizes of "updates" $|M_i|$ and the changes to the output. This reduces the cost of iterative computation of SSSP.

*(3) Assemble* simply takes $Q(G) = \bigcup_{i \in [1,n]} Q(F_i)$, the union of the shortest distance for each node in each $F_i$.

The process converges at correct $Q(G)$. Updates to $C_i.\bar{x}$ are "monotonic": the value of $\mathsf{dist}(s, v)$ for each node $v$ is computed from the active domain of $G$ and does not increase. Moreover, $\mathsf{dist}(s, v)$ is the shortest distance from $s$ to $v$ as warranted by the sequential algorithms [15, 28] (PEval and IncEval). □

**Fixpoint**. The GRAPE parallelization of the PIE program can be modeled as a simultaneous fixpoint operator $\phi(R_1, \ldots, R_m)$ defined on $m$ fragments. It starts with PEval for partial evaluation, and conducts incremental computation by taking with IncEval as the intermediate consequence operator, as follows:

$$\begin{aligned} R_i^0 &= \mathsf{PEval}(Q, F_i^0[\bar{x}_i]), \\ R_i^{r+1} &= \mathsf{IncEval}(Q, R_i^r, F_i^r[\bar{x}_i], M_i), \end{aligned}$$

where $i \in [1, m]$, $r$ indicates a superstep, $R_i^r$ denotes partial results in step $r$ at worker $P_i$, fragment $F_i^0 = F_i$, $F_i^r[\bar{x}_i]$ is fragment $F_i$ at the end of superstep $r$ carrying update parameters $C_i.\bar{x}$, and $M_i$ is a message indicating changes to $C_i.\bar{x}$. More specifically, (1) in the first superstep, PEval computes partial answers $R_i^0$ ($i \in [1, m]$). (2) At step $r + 1$, the partial answers $R_i^{r+1}$ are incrementally updated by IncEval, taking $Q$, $R_i^r$ and message $M_i$ as input. (3) The computation proceeds until $R_i^{r_0+1} = R_i^{r_0}$ at a fixpoint $r_0$ for all $i \in [1, m]$. At this point function Assemble is invoked to combine all partial answers $R_i^{r_0}$ and get the final answer $\rho(Q, G)$.

*Convergence*. The correctness of the fixpoint computation is characterized as follows. Given a graph computation problem $\mathcal{Q}$, (a) the sequential algorithm PEval for $\mathcal{Q}$ is correct if for all queries $Q \in \mathcal{Q}$ and graphs $G$, it terminates and returns $Q(G)$; (b) the sequential incremental algorithm IncEval for $\mathcal{Q}$ is *correct* if it correctly updates old output $Q(G)$ to $Q(G \oplus M)$, by computing the changes $\Delta O$ to be applied to $Q(G)$, for changes (messages) $M$ to update parameters; (c) Assemble is correct for $\mathcal{Q}$ *w.r.t.* partition strategy $\mathcal{P}$ if it correctly computes $Q(G)$ by assembling the partial answers from all workers, when GRAPE with PEval, IncEval and $\mathcal{P}$ terminates.

We say that GRAPE *correctly parallelizes* a PIE program $\rho$ with partition strategy $\mathcal{P}$ if for all $Q \in \mathcal{Q}$ and graphs $G$, GRAPE guarantees to reach a fixpoint such that $\rho(Q, G) = Q(G)$.

It is shown [14] that under BSP, GRAPE correctly parallelizes a PIE program $\rho$ for a graph computation problem $\mathcal{Q}$ with any partition strategy $\mathcal{P}$ if (a) PEval and IncEval of $\rho$ are correct sequential algorithms for $\mathcal{Q}$, and (b) Assemble correctly combines partial results, and (c) PEval and IncEval satisfy a monotonic condition. The condition is as follows: for all status variables $x \in C_i.\bar{x}$, $i \in [1, m]$, (a) the values of $x$ are from a finite set computed from the active domain of $G$ and (b) there exists a partial order $p_x$ on the values of $x$ such that IncEval updates $x$ in the order of $p_x$. That is, $x$ draws values from a finite domain (condition (a) above), and $x$ is updated "monotonically" following $p_x$ (condition (b)).

**Simulating other models**. The simple parallel model of GRAPE does not come with a price of degradation in the functionality. Following [33], we say that parallel model $\mathcal{M}_1$ can *optimally simulate* model $\mathcal{M}_2$ if there is a compilation algorithm that transforms any program with cost $C$ on $\mathcal{M}_2$ to a program with cost $O(C)$ on $\mathcal{M}_1$.

As shown in [14], GRAPE optimally simulates parallel models MapReduce [10], BSP [32] and PRAM (Parallel Random Access Machine) [33]. That is, all algorithms in these models with $n$ workers can be simulated by GRAPE using $n$ processors with the same number of supersteps and the same complexity. (2) We have shown that the simulation result above holds in the message-passing model described above, referred to as the designated message model in [14]. Hence, algorithms developed for graph systems based on MapReduce or BSP, *e.g.,* Pregel, GraphLab and Blogel, can be migrated to GRAPE without extra complexity.

**Features**. GRAPE has the following unique features.

(1) As shown in Fig. 4, to program with GRAPE, one only needs to provide a PIE program in the "plug" panel of GRAPE, which consists of (existing) sequential algorithms with minor changes. Given a partition strategy $\mathcal{P}$, a graph $G$, a query $Q$ and the number $m$ of processors in the "play" panel, GRAPE parallelizes the algorithms.

GRAPE aims to help users develop parallel programs, especially those who are more familiar with conventional sequential programming. This said, programming with GRAPE still requires to declare update parameters and design an aggregate function.

(2) Under a monotone condition, GRAPE parallelization guarantees to converge at the correct answer as long as the sequential algorithms are correct. This works regardless of partitioning strategy used, not limited to edge-cut and vertex-cut. Nonetheless, different strategies may yield partitions with various degrees of skewness and stragglers, which have an impact on the performance.

(3) GRAPE optimally simulates MapReduce, BSP and PRAM.

(4) GRAPE inherits existing optimization techniques developed for sequential graph algorithms, since it executes sequential algorithms on graph fragments, which are graphs themselves.

(5) GRAPE reduces the costs of iterative graph computations by using IncEval, to minimize unnecessary recomputations. While the speedup is more evident when IncEval is bounded [28], localizable or relatively bounded [11], these properties are not necessary.

There have been methods for incrementalizing graph algorithms, to get incremental algorithms from their batch counterparts [7].

## 3. PROGRAMMING WITH GRAPE

We next outline PIE programs for graph pattern matching (Sim), connectivity (CC) and collaborative filtering (CF), under edge-cut. PIE programs under vertex-cut can be developed similarly.

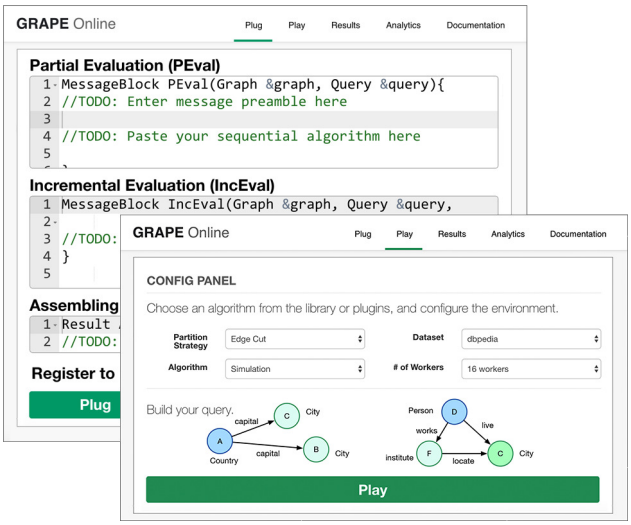


Figure 4: Programming Interface of GRAPE

**Graph simulation** (Sim). A *graph pattern* is a graph $Q = (V_Q, E_Q, L_Q)$, in which (a) $V_Q$ is a set of *query nodes*, (b) $E_Q$ is a set of *query edges*, and (c) each node $u$ in $V_Q$ carries a label $L_Q(u)$.

A graph $G$ *matches* a pattern $Q$ via *simulation* if there is a binary relation $R \subseteq V_Q \times V$ such that (a) for each query node $u \in V_Q$, there exists a node $v \in V$ such that $(u, v) \in R$, and (b) for each pair $(u, v) \in R$, $L_Q(u) = L(v)$, and for each query edge $(u, u')$ in $E_Q$, there exists an edge $(v, v')$ in graph $G$ such that $(u', v') \in R$.

It is known that if $G$ matches $Q$, then there exists a *unique maximum* relation [20], referred to as $Q(G)$. If $G$ does not match $Q$, $Q(G)$ is the empty set. Given a directed graph $G$ and a pattern $Q$, graph simulation is to compute the maximum relation $Q(G)$.

We show how GRAPE parallelizes graph simulation.

*(1)* PEval. GRAPE takes the sequential simulation algorithm of [20] as PEval to compute $Q(F_i)$ in parallel. PEval declares a Boolean status variable $x_{(u,v)}$ for each node $u$ in $V_Q$ and each node $v$ in fragment $F_i$, indicating whether $v$ matches $u$, initialized true. It takes $F_i.I$ as candidate set $C_i$. For each node $u \in V_Q$, PEval computes a set $\mathsf{sim}(u)$ of candidate matches $v$ in $F_i$, and iteratively removes from $\mathsf{sim}(u)$ those nodes that violate the simulation condition (see [20] for details). At the end of the process, PEval sends $C_i.\bar{x} = \{x_{(u,v)} \mid u \in V_Q, v \in F_i.I\}$ to master $P_0$.

At master $P_0$, GRAPE maintains $x_{(u,v)}$ for all $v \in \mathcal{F}.I$. Upon receiving messages from all workers, it changes $x_{(u,v)}$ to false if it is false in *one of* the messages. This is specified by min as $f_{\text{aggr}}$, taking the order false $\prec$ true. GRAPE finds those variables that become false, groups them into messages $M_j$, and sends $M_j$ to $P_j$.

*(2)* IncEval is the sequential incremental graph simulation algorithm of [12] in response to edge deletions. If $x_{(u,v)}$ is changed to false by message $M_i$, it is treated as deletion of "cross edges" to $v \in F_i.O$. It starts with changed status variables in $M_i$, propagates the changes to affected area, and removes from sim matches that become invalid (see [12] for details). The partial result is now the revised sim relation. At the end of the process, IncEval sends to $P_0$ updated values of status variables in $C_i.\bar{x}$, as in PEval.

IncEval is *semi-bounded* [12]: its cost is decided by the sizes of "updates" $|M_i|$ and changes to the affected area necessarily checked by all incremental algorithms for Sim, not by $|F_i|$.

*(3)* Assemble simply takes $Q(G) = \bigcup_{i \in [1,n]} Q(F_i)$, the union of all partial matches, *i.e.,* relation sim at each fragment $F_i$.

*(4) Correctness* is warranted by the convergence condition of GRAPE, as the sequential algorithms [12, 20] (PEval and IncEval)

are correct, and updates to $C_i.\bar{x}$ are monotonic: $x_{(u,v)}$ is initially true for each border node $v$, and is changed at most once to false.

**Graph connectivity** (CC). Given an undirected graph $G$, CC computes all connected components of $G$, referred to as *CC*s.

*(1)* PEval declares an integer variable $v.\mathsf{cid}$ for each node $v$ in fragment $F_i$, initialized as its node id. It uses a standard sequential traversal (*e.g.,* DFS) to compute the local *CC*s of $F_i$ and determines $v.\mathsf{cid}$ for each $v \in F_i$. For each local *CC* $C$, (a) PEval creates a "root" node $v_c$ carrying the minimum node id in $C$ as $v_c.\mathsf{cid}$, and (b) links all the nodes in $C$ to $v_c$, and sets their cid as $v_c.\mathsf{cid}$. These can be completed in one pass of the edges of $F_i$ via DFS. At the end of process, PEval sends $\{v.\mathsf{cid} \mid v \in F_i.I\}$ to master $P_0$.

At master $P_0$, GRAPE maintains $v.\mathsf{cid}$ for each all $v \in \mathcal{F}.I$. It updates $v.\mathsf{cid}$ by taking the smallest cid if multiple cids are received, by taking min as $f_{\mathsf{aggr}}$ in PEval. It groups the border nodes with updated cids into messages $M_j$, and sends $M_j$ to $P_j$.

*(2)* IncEval incrementally updates the cids of the nodes in $F_i$ upon receiving $M_i$. The message $M_i$ sent to $P_i$ consists of $v.\mathsf{cid}$ with updated (smaller) values of its border nodes $v$. For each $v$ in $M_i$, IncEval (a) finds the root $v_c$ of $v$, and (b) for $v_c$ and all the border nodes linked to it, directly changes their cids to $v.\mathsf{cid}$.

Note that IncEval is *bounded*: it takes $O(|M_i|)$ time to identify the root nodes, and $O(|\mathsf{AFF}|)$ time to update cids by following the direct links from the root nodes, where AFF consists of only those nodes with their cid *changed*, independent of $|F_i|$.

*(3)* Assemble first updates the cid of each node to the cid of its linked root node. It then merges all the nodes having the same cids in a single bucket, and returns all buckets as *CC*s.

*(4) Correctness*. It is easy to see that the process terminates since the cids of the nodes are monotonically decreasing by aggregate function $f_{\mathsf{aggr}}$ until no changes can be made. Moreover, it correctly merges two local *CC*s by propagating smaller component ids.

**Collaborative filtering** (CF). CF takes as input a bipartite graph $G$ that includes users $U$ and products $P$, and a set of weighted edges $E \subseteq U \times P$ [23]. (1) Each user $u \in U$ (resp. product $p \in P$) carries latent factor vector $u.f$ (resp. $p.f$). (2) Each edge $e = (u, p)$ in $E$ carries a weight $r(e)$, estimated as $u.f^T * p.f$ ($\emptyset$ for "unknown") that encodes a rating from user $u$ to product $p$. The *training set* $E_T$ refers to edge set $\{e \mid r(e) \neq \emptyset, e \in E\}$, *i.e.,* all the known ratings. Given these, CF computes the missing factor vectors $u.f$ and $p.f$ to minimize an error function $\epsilon(f, E_T) = \mathsf{min} \sum_{((u,p)\in E_T)} (r(u,p) - u.f^T p.f)^2 + \lambda(\|u.f\|^2 + \|p.f\|^2)$. This is typically carried out by the stochastic gradient descent (SGD) algorithm [23], which iteratively (1) predicts error $\epsilon(u,p) = r(u,p) - u.f^T * p.f$, for each $e = (u,p) \in E_T$, and (2) updates $u.f$ and $p.f$ accordingly to minimize $\epsilon(f, E_T)$.

GRAPE parallelizes CF by adopting SGD [23] as PEval, and the incremental algorithm ISGD of [34] as IncEval, using master $P_0$ to synchronize the shared factor vectors $u.f$ and $p.f$.

*(1)* PEval. It sets $C_i = F_i.I$ and declares status variable $v.x = (v.f, t)$ for $v \in C_i$, where $v.f$ is the factor vector of $v$ (initially $\emptyset$), and $t$ bookkeeps a timestamp at which $v.f$ is lastly updated. PEval is essentially the sequential SGD of [23]. It processes a "mini-batch" of training examples independently of others, to compute prediction error $\epsilon(u,p)$, and updates factor vectors $f$ by a magnitude proportional to $\gamma$ in the opposite direction of the gradient as:

$$u.f^t = u.f^{t-1} + \gamma(\epsilon(u,p) * v.f^{t-1} - \lambda * u.f^{t-1}); \quad (1)$$

$$p.f^t = p.f^{t-1} + \gamma(\epsilon(u,p) * u.f^{t-1} - \lambda * p.f^{t-1}). \quad (2)$$

At the end of its process, PEval sends messages $M_i$ that consists of updated $v.x$ for each $v \in C_i = F_i.O$ to master $P_0$.

At $P_0$, GRAPE maintains $v.x = (v.f, t)$ for all border nodes $v \in \mathcal{F}.I = \mathcal{F}.O$. Upon receiving updated values $(v.f', t')$ with $t' > t$, it changes $v.f$ to $v.f'$, *i.e.,* it takes max as aggregate function $f_{\mathsf{aggr}}$ on timestamps. GRAPE then groups the updated vectors into messages $M_j$, and sends $M_j$ to $P_j$ as usual.

*(2)* IncEval is the incremental algorithm ISGD of [34]. Upon receiving message $M_i$ at worker $P_i$, it computes $F_i \oplus M_i$ by treating $M_j$ as updates to factor vectors of nodes in $F_i.I$, and only modifies affected factor vectors as in PEval based solely on new observations. It sends the updated vectors in $C_i$ as in PEval.

*(3)* Assemble simply takes the union of all the factor vectors of nodes from the workers (to be used for recommendation).

*(4) Correctness*. The convergence condition in a sequential SGD algorithm [23, 34] is specified either as a predetermined maximum number of supersteps (*e.g.,*GraphLab), or when $\epsilon(f, E_T)$ is smaller than a threshold. In either case, GRAPE correctly infers CF models guaranteed by the correctness of SGD and ISGD, and by monotonic updates with the latest changes as in sequential SGD algorithms.

# 4. PERFORMANCE STUDY

We have implemented GRAPE [13]. We next empirically evaluate its efficiency and communication cost, using real-life and synthetic graphs. We compared the performance of GRAPE with three systems: Giraph (an open-source version of Pregel), GraphLab, and Blogel (the fastest block-centric system we are aware of).

**Experimental setting**. We used five real-life graphs of different types, including (1) traffic [5], an (undirected) US road network with 23 million nodes (locations) and 58 million edges; (2) UKWeb [6], a large Web graph with 133 million nodes and 5 billion edges; (3) Friendster [2], a social network with 65 million users and 1.8 billion relations; (4) DBpedia [1], a knowledge base with 5 million entities and 54 million edges, and in total 411 distinct labels; and (5) movieLens [4], a dense recommendation network (bipartite graph) with 20 million movie ratings (as weighted edges) between a set of 138000 users and 27000 movies. To test Sim with unlabeled Friendster, we generated 100 random node labels. We also randomly assigned weights to all graphs for testing SSSP.

*Queries*. We randomly generated the following queries. (a) We sampled 10 source nodes in each graph, and constructed an SSSP query for each node. (b) We generated 20 pattern queries for Sim, controlled by $|Q| = (|V_Q|, |E_Q|)$, the number of nodes and edges, respectively, using labels drawn from the graphs.

We remark that GRAPE can process query load without reloading the graph, but GraphLab, Giraph and Blogel need to reload the graph each time a query is issued, which is costly over large graphs.

*Algorithms*. We implemented the PIE programs for those query classes given in Sections 2 and 3. We used XtraPuLP [30] as the default graph partition strategy. We adopted basic sequential algorithms for all the systems without further optimization.

We also implemented algorithms for the queries for Giraph, GraphLab and Blogel. We used "default" code provided by the systems when available, and made our best efforts to develop "optimal" algorithms otherwise (see [14] for more details). We implemented synchronized algorithms for both GraphLab and Giraph for the ease of comparison. We expect the observed relative performance trends to hold on other similar graph systems.

We deployed the systems on a cluster of up to 12 machines, each with 16 threads of Intel Xeon 2.2GHz, and 128G memory. On each thread, a worker is deployed (thus in total 192 workers). Each experiment was run 5 times and the average is reported here.

**Experimental results**. We next report our findings.

**Exp-1: Efficiency**. We first evaluated the efficiency of GRAPE by varying the number $n$ of workers used, from 64 to 192. For SSSP and CC, we experimented with UKWeb, traffic and Friendster. For Sim, we used over Friendster and DBpedia. We used movieLens for CF as its application in movie recommendation.

*(1)* SSSP. Figures 5a-5c report the performance of the four systems for SSSP over traffic, UKWeb and Friendster, respectively. From the results we can see the following.

(a) GRAPE outperforms Giraph, GraphLab and Blogel by 14842, 3992 and 756 times, respectively, over traffic with 192 workers (Fig 5a). In the same setting, it is 556, 102 and 36 times faster over UKWeb (Fig. 5b), and 18, 1.7 and 4.6 times faster over Friendster (Fig. 5c). These tell us that by simply parallelizing sequential algorithms without further optimization, GRAPE already outperforms the state-of-the-art systems in response time.

The improvement of GRAPE over all the systems on traffic is much larger than on Friendster and UKWeb. (i) For Giraph and GraphLab, this is because synchronous vertex-centric algorithms take more supersteps to converge on graphs with larger diameters, *e.g.,* traffic. With 192 workers, Giraph take 10749 supersteps over traffic and 161 over UKWeb; similarly for GraphLab. In contrast, GRAPE is not vertex-centric and it takes 31 supersteps on traffic and 24 on UKWeb. (ii) Blogel also takes more (1690) supersteps over traffic than over UKWeb (42) and Friendster (23). It generates more blocks over traffic (with larger diameter) than UKWeb and Friendster. Since Blogel treats blocks as vertices, the benefit of parallelism is degraded with more blocks. (iii) GRAPE reduces redundant computation by the use of incremental IncEval.

(b) In all cases, GRAPE takes less time when $n$ increases. On average, it is 1.4, 2.3 and 1.5 times faster for $n$ from 64 to 192 over traffic, UKWeb and Friendster, respectively. (i) Compared with the results in [14] using less workers, this improvement degrades a bit. This is mainly because the larger number of fragments leads to more communication overhead. On the other hand, such impact is significantly mitigated by IncEval that only ships changed update parameters. (ii) In contrast, Blogel does not demonstrate such consistency in scalability. It takes more time on traffic when $n$ is larger. When $n$ varies from 160 to 192, it takes longer over Friendster. Its communication cost dominates the parallel cost as $n$ grows, "canceling out" the benefit of parallelism. (iii) GRAPE has scalability comparable to GraphLab over Friendster and scales better over UKWeb and traffic. Giraph has better improvement with larger $n$, but with constantly higher cost (see (a)) than GRAPE.

(c) GRAPE significantly reduces supersteps. It takes on average 22 supersteps, while Giraph, GraphLab and Blogel take 3647, 3647 and 585 supersteps, respectively. This is because GRAPE runs sequential algorithms over fragmented graphs with cross-fragment communication only when necessary, and IncEval ships only *changes* to status variables. In contrast, Giraph, GraphLab and Blogel pass vertex-vertex (vertex-block) messages.

*(2)* CC. Figures 5d-5f report the performance for CC, and tell us the following. (a) Both GRAPE and Blogel substantially outperform Giraph and GraphLab. For instance, when $n = 192$, GRAPE is on average 12094 and 1329 times faster than Giraph and GraphLab, respectively. (b) Blogel is faster than GRAPE in some cases, *e.g.,* 3.5s vs. 17.9s over UKWeb when $n = 192$. This is because Blogel embeds the computation of CC in its graph partition phase as precomputation, while this graph partition cost (on average 357 seconds using its built-in Voronoi partition) is *not* included in its response time. In other words, without taking advantage of precomputation, the performance of GRAPE is already comparable to the near "optimal" case reported by Blogel.

*(3)* Sim. Fixing $|Q| = (6, 10)$, *i.e.,* patterns $Q$ with 6 nodes and 10 edges, we evaluated graph simulation over DBpedia and Friendster. As shown in Figures 5g-5h, (a) GRAPE consistently outperforms Giraph, GraphLab and Blogel over all queries. It is 109, 8.3 and 45.2 times faster over Friendster, and 136.7, 5.8 and 20.8 times faster over DBpedia on average, respectively, when $n = 192$. (b) GRAPE scales better with the number $n$ of workers than the others. (c) GRAPE takes at most 21 supersteps, while Giraph, GraphLab and Blogel take 38, 38 and 40 supersteps, respectively. This empirically validates the convergence guarantee of GRAPE under monotonic status-variable updates and its positive effect on reducing parallel and communication cost.

*(4) Collaborative filtering (*CF*).* We used movieLens [4] with a training set $|E_T| = 90\%|E|$. We compared GRAPE with the built-in CF code in GraphLab, and with CF programs implemented for Giraph and Blogel. Note that CF favors "vertex-centric" programming since each node only needs to exchange data with their neighbors, as indicated by that GraphLab and Giraph outperform Blogel. Nonetheless, Figure 5i shows that GRAPE is on average 4.1, 2.6 and 12.4 times faster than Giraph, GraphLab and Blogel, respectively. Moreover, it scales well with $n$.

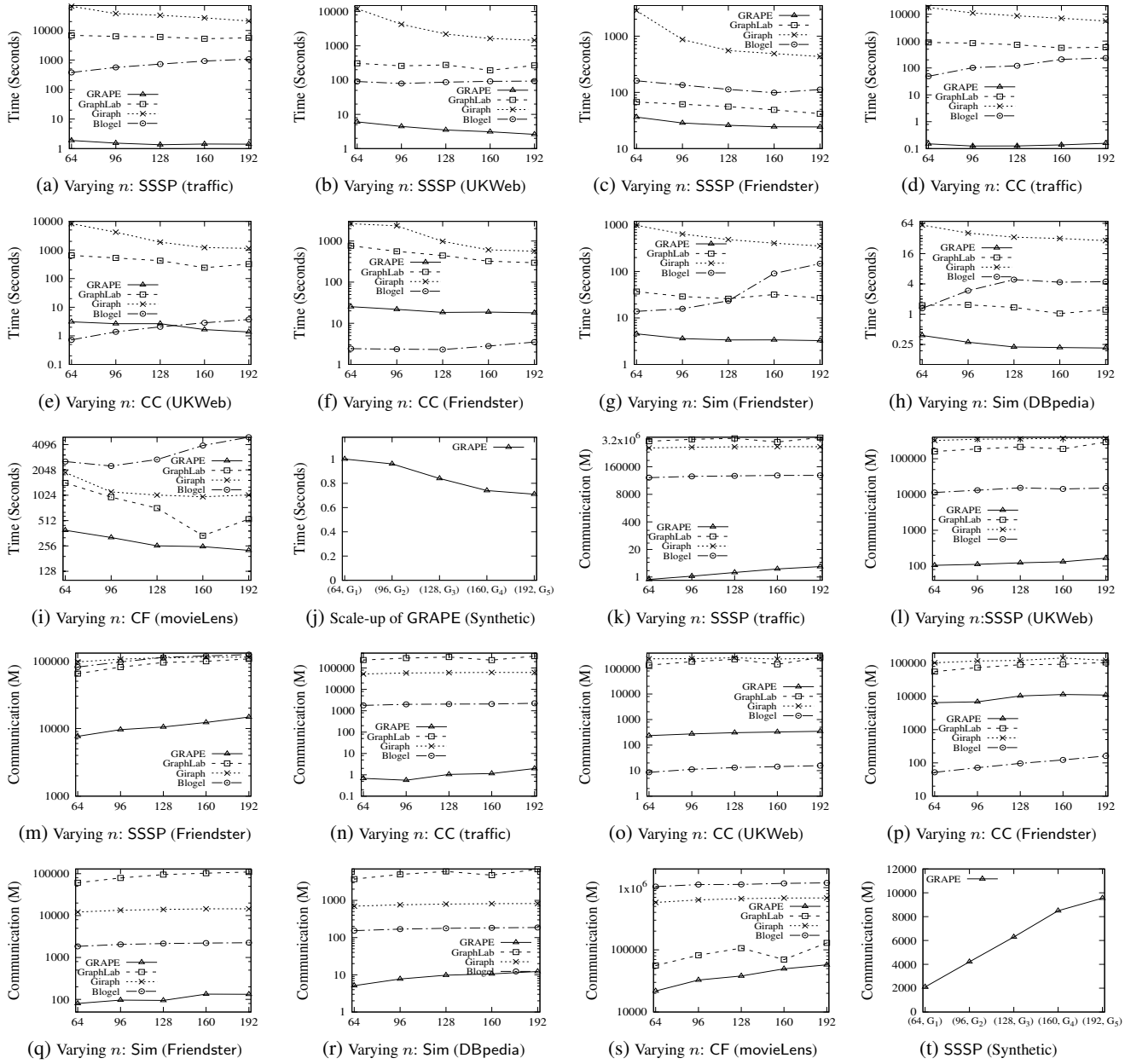
*(5) Scale-up of* GRAPE. The speed-up of a system may degrade over more workers [26]. We thus evaluate the scale-up of GRAPE, which measures the ability to keep the same performance when both the size of graph $G$ (denoted as $(|V|, |E|)$) and the number $n$ of workers increase proportionally. We varied $n$ from 64 to 192, and for each $n$, deployed GRAPE over a synthetic graph. The graph size varies from $(50M, 500M)$ to $(250M, 2.5B)$ (denoted as $G_5$), with fixed ratio between edge number and node number and proportional to $n$ . The scale up at *e.g.,* $(128, G_3)$ is the ratio of the time using 64 workers over $G_1$ to its counterpart using 128 workers over $G_3$. As shown in Fig. 5j, GRAPE preserves a reasonable scale-up (close to linear scale-up, the optimal scale-up).

Compared to single-threaded computation, GRAPE incurs extra communication overhead, just like other parallel systems. However, large graphs such as UKWeb are beyond the capacity of a single machine, and parallel computation is a must for such graphs.

**Exp-2: Communication cost**. The communication cost (in bytes) reported by Giraph, GraphLab and Blogel depends on their own implementation of message blocks and protocols [19]. For a fair comparison, we tracked the total bytes sent by each machine during a run, by monitoring the system file /proc/net/dev, following [19].

In the same setting as Exp-1, Figures 5l-5t report the communication costs of the systems. We observe that Giraph and GraphLab ship roughly the same amount of messages. GRAPE ships much less data than Giraph and GraphLab. On datasets excluding traffic, with 192 workers, it ships on average 0.095%, 0.62%, 0.3%, and 26.2% of the data shipped for SSSP, Sim, CC and CF by Giraph (GraphLab), respectively, and reduces cost up to 6 orders of magnitude on traffic! While it ships more data than Blogel for CC due to the precomputation of Blogel, it only ships 1.9%, 6.2% and 4.8% of the data shipped by Blogel for SSSP, Sim and CF, respectively.

*(1)* SSSP. Figures 5k-5m show that both GRAPE and Blogel incur communication costs that are orders of magnitudes less than those of GraphLab and Giraph. This is because vertex-centric programming incurs a large number of inter-vertex messages. Both block-centric programs (Blogel) and PIE programs (GRAPE) effectively




(a) Varying $n$: SSSP (traffic) (b) Varying $n$: SSSP (UKWeb) (c) Varying $n$: SSSP (Friendster) (d) Varying $n$: CC (traffic)

(e) Varying $n$: CC (UKWeb) (f) Varying $n$: CC (Friendster) (g) Varying $n$: Sim (Friendster) (h) Varying $n$: Sim (DBpedia)

(i) Varying $n$: CF (movieLens) (j) Scale-up of GRAPE (Synthetic) (k) Varying $n$: SSSP (traffic) (l) Varying $n$:SSSP (UKWeb)

(m) Varying $n$: SSSP (Friendster) (n) Varying $n$: CC (traffic) (o) Varying $n$: CC (UKWeb) (p) Varying $n$: CC (Friendster)

(q) Varying $n$: Sim (Friendster) (r) Varying $n$: Sim (DBpedia) (s) Varying $n$: CF (movieLens) (t) SSSP (Synthetic)

Figure 5: Efficiency and communication cost of GRAPE

reduce unnecessary messages, and trigger inter-block messages only when necessary. Moreover, GRAPE ships $0.9\%$ and $10\%$ of the data shipped by Blogel over UKWeb and Friendster, respectively. Indeed, GRAPE ships only updated values. This significantly reduces the amount of messages that need to be shipped.

*(2)* CC. Figures 5n-5p show similar improvement of GRAPE over GraphLab and Giraph. It ships on average $0.17\%$ of the data shipped by Giraph and GraphLab. As Blogel precomputes CC (see Exp-1(2)), it ships little data. This said, GRAPE is not far worse than the near "optimal" case of Blogel, sometimes better.

*(3)* Sim. Figures 5q and 5r report the communication cost for graph simulation over Friendster and DBpedia, respectively. One can see that GRAPE ships substantially less data, *e.g.,* on average $0.9\%$, $0.1\%$ and $4.9\%$ of the data shipped by Giraph, GraphLab and Blogel, respectively. Observe that the communication cost of Blogel is much higher than that of GRAPE, even though it adopts inter-block communication. This shows that the extension of vertex-centric to block-centric by Blogel has limited improvement for more complex queries. GRAPE works better than these systems by employing incremental IncEval to reduce excessive messages.

*(4)* CF. Figure 5s reports the result for CF over movieLens. On average, GRAPE ships $5.6\%$, $43.3\%$ and $3.2\%$ of the data shipped by Giraph, GraphLab and Blogel, respectively.

*(5) Communication cost (synthetic).* In the same setting as Figure 5j, Figure 5t reports the communication cost for SSSP over large synthetic graphs. It takes higher cost over larger graphs and more workers due to increased "border nodes", as expected. The results for other algorithms are consistent and hence not shown.

**Summary**. We find the following. (1) Over traffic [5], GRAPE is on average 4, 3 and 2 orders of magnitude faster than Giraph, GraphLab and Blogel for SSSP, respectively, with 192 processors,

due to the large diameter of the graph. On other real-life graphs excluding traffic, GRAPE is on average 484, 36 and 15 times faster than the three systems for SSSP, 151, 6.8 and 16 times for Sim, and 4.6, 2.6 and 12.4 times for CF, respectively, when the number of workers ranges from 64 to 192. For CC, it is 1377 and 212 times faster than Giraph and GraphLab, respectively, and is comparable to the "optimal" case of Blogel. (2) In the same setting (excluding traffic), GRAPE ships on average 0.07%, 0.12% and 1.7% of the data shipped across machines by Giraph, GraphLab and Blogel for SSSP, 0.89%, 0.14% and 4.9% for Sim, 5.6%, 43.3% and 3.2% for CF, respectively. When traffic is also included, GRAPE outperforms these systems by up to 6 orders of magnitude in communication cost for SSSP. For CC, it incurs 0.23% and 0.3% of data shipment of Giraph and GraphLab, and is comparable with "optimized" Blogel. (3) GRAPE demonstrates good scale-up when using more workers, since its incremental computation mitigates the impact of more border nodes and fragments. Moreover, incremental steps effectively reduce unnecessary recomputation.

# 5. CONCLUSION

The main objective of GRAPE is to simplify parallel programming for graph computations, from "think parallel" to "think sequential". For users who are used to conventional programming, they can start with (existing) sequential algorithms, add declarations for handling messages, and let GRAPE parallelize the computation across a cluster of machines. Moreover, GRAPE guarantees to converge at correct answers under a general condition as long as it is provided with correct sequential algorithms, and it inherits optimization strategies developed for sequential graph algorithms.

As proof of concept (PoC), we have deployed and evaluated GRAPE at three companies. At a large online payment company, GRAPE serves as the graph computing infrastructure supporting its financial risk control system. The company employs graphs in which vertices denote customers, and edges represent transactions and associations with other customers; it needs to evaluate the customers and assign a credit. The company used to deploy its system on Neo4j + Hive + Spark. However, none of the systems can process the tasks alone; the workflow spans three systems and takes 15 minutes on average for a single query. In contrast, GRAPE provides a unified solution for this scenario. It supports real-time ad-hoc queries without the need to couple with other systems. It improves the performance of financial risk analyses: it is 9.0 times faster in graph batch ingesting and streaming, 128.8 times faster in association analysis, and is faster by up to 5 orders of magnitude in batch processing of real-life business applications.

GRAPE works well for other applications. We have also carried out PoC at a big-data service company and a telecommunication service company. The results are consistent and very promising.

We are currently extending GRAPE to support a new parallel model that adaptively switches between synchronous and asynchronous models, to reduce stragglers and stale computations.

**Acknowledgments**. Fan, Cao, Xu and Yu are supported in part by 973 Program 2014CB340302, ERC 652976, EPSRC EP/M025268/1, NSFC 61421003 and 61602023, and Beijing Advanced Innovation Center for Big Data and Brain Computing. Wu is supported in part by NSF IIS-1633629.

# 6. REFERENCES

[1] DBpedia. *http://wiki.dbpedia.org/Datasets.*
[2] Friendster. *https://snap.stanford.edu/data/com-Friendster.html.*
[3] Giraph. *http://giraph.apache.org/.*
[4] Movielens. *http://grouplens.org/datasets/movielens/.*
[5] Traffic. *http://www.dis.uniroma1.it/challenge9/download.shtml.*
[6] UKWeb. *http://law.di.unimi.it/webdata/uk-union-2006-06-2007-05/*, 2006.
[7] U. A. Acar. *Self-Adjusting Computation*. PhD thesis, CMU, 2005.
[8] K. Andreev and H. Racke. Balanced graph partitioning. *Theory of Computing Systems*, 39(6):929–939, 2006.
[9] F. Bourse, M. Lelarge, and M. Vojnovic. Balanced graph edge partition. In *SIGKDD*, pages 1456–1465, 2014.
[10] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1), 2008.
[11] W. Fan, C. Hu, and C. Tian. Incremental graph computations: Doable and undoable. In *SIGMOD*, 2017.
[12] W. Fan, X. Wang, and Y. Wu. Incremental graph pattern matching. *TODS*, 38(3), 2013.
[13] W. Fan, J. Xu, Y. Wu, W. Yu, and J. Jiang. GRAPE: Parallelizing sequential graph computations. *PVLDB*, 10(12):1889–1892, 2017.
[14] W. Fan, J. Xu, Y. Wu, W. Yu, J. Jiang, B. Zhang, Z. Zheng, Y. Cao, and C. Tian. Parallelizing sequential graph computations. In *SIGMOD*, 2017.
[15] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *JACM*, 34(3), 1987.
[16] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *USENIX*, 2012.
[17] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. GraphX: Graph processing in a distributed dataflow framework. In *OSDI*, 2014.
[18] I. Grujic, S. Bogdanovic-Dinic, and L. Stoimenov. Collecting and analyzing data from E-Government Facebook pages. In *ICT Innovations*, 2014.
[19] M. Han, K. Daudjee, K. Ammar, M. T. Ozsu, X. Wang, and T. Jin. An experimental comparison of Pregel-like graph processing systems. *VLDB*, 7(12), 2014.
[20] M. R. Henzinger, T. Henzinger, and P. Kopke. Computing simulations on finite and infinite graphs. In *FOCS*, 1995.
[21] N. D. Jones. An introduction to partial evaluation. *ACM Computing Surveys*, 28(3), 1996.
[22] M. Kim and K. S. Candan. SBV-Cut: Vertex-cut based graph partitioning using structural balance vertices. *Data & Knowledge Engineering*, 72:285–303, 2012.
[23] Y. Koren, R. Bell, and C. Volinsky. Matrix factorization techniques for recommender systems. *IEEE Computer*, 42(8):30–37, 2009.
[24] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Distributed GraphLab: A framework for machine learning in the cloud. *PVLDB*, 5(8), 2012.
[25] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *SIGMOD*, 2010.
[26] F. McSherry, M. Isard, and D. G. Murray. Scalability! But at what cost? In *HotOS*, 2015.
[27] G. Ramalingam and T. Reps. An incremental algorithm for a generalization of the shortest-path problem. *J. Algorithms*, 21(2):267–305, 1996.
[28] G. Ramalingam and T. Reps. On the computational complexity of dynamic graph problems. *TCS*, 158(1-2), 1996.
[29] B. Shao, H. Wang, and Y. Li. Trinity: A distributed graph engine on a memory cloud. In *SIGMOD*, 2013.
[30] G. M. Slota, S. Rajamanickam, K. Devine, and K. Madduri. Partitioning trillion-edge graphs in minutes. In *IPDPS*, 2017.
[31] Y. Tian, A. Balmin, S. A. Corsten, and J. M. Shirish Tatikonda. From "think like a vertex" to "think like a graph". *PVLDB*, 7(7), 2013.
[32] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.
[33] L. G. Valiant. General purpose parallel architectures. In *Handbook of Theoretical Computer Science, Vol A*. 1990.
[34] J. Vinagre, A. M. Jorge, and J. Gama. Fast incremental matrix factorization for recommendation with positive-only feedback. In *UMAP*, 2014.
[35] G. Wang, W. Xie, A. J. Demers, and J. Gehrke. Asynchronous large-scale graph processing made easy. In *CIDR*, 2013.
[36] D. Yan, Y. Bu, Y. Tian, and A. Deshpande. Big graph analytics platforms. *Foundations and Trends in Databases*, 7(1-2), 2017.
[37] D. Yan, J. Cheng, Y. Lu, and W. Ng. Blogel: A block-centric framework for distributed computation on real-world graphs. *PVLDB*, 7(14):1981–1992, 2014.

# Technical Perspective: Supporting Linear Algebra Operations in SQL

Yannis Papakonstantinou
Computer Science and Engineering Department
University of California, San Diego
yannis@cs.ucsd.edu

Linear algebra operations are at the core of Machine Learning. Multiple specialized systems have emerged for the scalable, distributed execution of matrix and vector operations. The relationship of such computations to data management and databases however brings frictions. It is well known that a great deal of human time and machine time is being spent nowadays on fetching data out of the database and performing a computation on a specialized system. One answer to the issue is that we truly need a new kind of non-SQL database that is tuned to these computations.

The creators of SimSQL opted for the decidedly incremental approach. Can we make a very small set of changes to the relational model and RDBMS software to render them suitable for executing linear algebra in the database?

We have come across the "brand new system" versus "incremental to relational" question many times in the database field. E.g., do we need brand new query languages and query processors for data cubes? Or do we need to have our query processors pay attention to specific cases that are especially common in data analytics queries over stars and snowflakes? Do semistructured query languages need to depart from SQL or it is enough to be incremental to SQL? Same for query processors. Repeat the questions to graph data and RDF data. In many cases, new custom systems emerged only to figure out later that we could/should have tackled the problem incrementally. That's the trap that the authors of this paper avoid.

This is not to say that radical changes and extensions should be forbidden. Rather it says that we should closely scrutinize the necessity of the changes, do them when needed and keep them minimal. The authors identify the right opportunities. Here is a non-exhaustive list:

(a) Writing matrix and vector operations as a join over the index can be syntactically tedious. They solve the problem by introducing special syntactic features.

(b) They notice a connection between signatures and size estimation and exploit it.

(c) They allow their query user to move across different denormalizations to find the one that makes sense from expressiveness and performance point of view. The point where types relate to performance is whether the right level of granularity for distribution in a shared-nothing architecture is specified.

Overall, the extensions of the paper follow a thoughtful and minimal approach that is worth studying in the particular field of linear algebra operations, as well as generally in the design of systems for analytics.

# Scalable Linear Algebra on a Relational Database System

Shangyu Luo
Rice University
sl45@rice.edu

Zekai J. Gao
Rice University
jacobgao@rice.edu

Michael Gubanov
U. of Texas, San Antonio
mikhail.gubanov@utsa.edu

Luis L. Perez
Rice University
lperezp@gmail.com

Christopher Jermaine
Rice University
cmj4@rice.edu

## ABSTRACT

Scalable linear algebra is important for analytics and machine learning (including deep learning). In this paper, we argue that a parallel or distributed database system is actually an excellent platform upon which to build such functionality. Most relational systems already have support for cost-based optimization—which is vital to scaling linear algebra computations—and it is well-known how to make relational systems scale. We show that by making just a few changes to a parallel/distributed relational database system, such a system can be a competitive platform for scalable linear algebra. Our results suggest that brand new systems supporting scalable linear algebra are not absolutely necessary, and that such systems could instead be built on top of existing relational technology.

## 1. INTRODUCTION

To support machine learning and large-scale statistical processing, a new category of data processing system has appeared: the scalable linear algebra system. Unlike established, long-lived efforts aimed at building scalable linear algebra APIs (such as ScaLAPACK [7]), these newer efforts are targeted more towards building complete data management systems that support storage/retrieval of data to/from disk, buffering/caching of data, and automatic logical/physical optimizations of computations (automatic re-writing of queries, pipelining, etc.). Such systems may also offer some form of recovery, as well as offering a special-purpose domain-specific language. For example, SystemML, developed at IBM [16], as well as RIOT [25] and Cumulon [18] provide scalable linear algebra capabilities as well as many features borrowed from data management systems. Big Data systems typically provide linear algebra APIs (such as Spark's `mllib.linalg` [1]). Modern array database systems such as SciDB [11] also offer direct support for linear algebra.

**Is a New Type of System Actually Necessary?** While supporting scalable linear algebra in the context of a full-fledged data management system is clearly a desirable goal, the hypothesis underlying this paper is that with just a few changes, a classical, parallel relational database is actually an excellent platform for building a scalable linear algebra system.

In practice, many (or even most) distributed linear algebra computations have closely corresponding, distributed relational algebra computations. Given this, we believe that it is natural to build distributed linear algebra functionality on top of a distributed relational database system. Such systems are highly performant, reaping the benefits of decades of research and engineering effort targeted at building efficient systems. Further, relational systems already have software components such as a cost-based query optimizer to aid in performing efficient computations. In fact, much of the work that goes into developing a scalable linear algebra system from the ground up [9] requires implementing functionality that looks a lot like a database query optimizer [14].

Given that much of the world's data currently sits in relational databases, and that dataflow systems increasingly provide at least some support for relational processing [5, 23], building linear algebra support into relational systems would mean that much of the world's data would be sitting in systems capable of performing scalable linear algebra. This would have several obvious benefits:

1. It would eliminate the "extract-transform-reload nightmare", particularly if the goal is performing analytics on data already stored in a relational system.
2. It would obviate the need for practitioners to adopt yet another type of data processing system in order to perform mathematical computations.
3. The design and implementation of high-performance distributed and parallel relational systems is well-understood. If it is possible to adapt such a system to the task of scalable linear algebra, most or all of the science and engineering performed over decades, aimed at determining how to build a distributed relational system, is directly applicable.

**Towards in-database linear algebra.** We ask the question:

*Can we make a very small set of changes to the relational model and a RDBMS software to render them suitable for in-database linear algebra?*

The approach we examine is actually simple: we consider adding new `LABELED_SCALAR`, `VECTOR`, and `MATRIX` data types to an SQL-based relational system. This facilitates efficient, distributed linear algebra operations in SQL. Technically, this seems to be a rather minor change. After all, `array` has been available as a data type in most modern DBMSs—arrays can clearly be used to encode vectors and matrices—and some database systems (such as Oracle) offer a form of integration between arrays and linear algebra libraries such as BLAS [8] and LAPACK [4]. However, these previous, ad-hoc approaches do not offer complete integration with the database system. The query optimizer, for example, does not understand the semantics of calls to linear algebra operations, and this results in lost opportunities for optimization. Thus, we also


DOI: 10.1109/ICDE.2017.108.

consider a small set of changes to a relational query optimizer that can render it somewhat "linear algebra aware".

There are clearly drawbacks to our minimalist approach. Compared to systems such as SystemML and Riot, which offer higher-level, non-SQL programming abstractions, a programmer's intent may be obfuscated by using an extended SQL. For example, an optimizer implemented by our approach may be unable to optimize the order of a chain of distributed matrix multiplies expressed in SQL. Further, a programmer using our extensions to implement distributed matrix operations must make key choices regarding the blocking or chunking of the matrices.

Still, we believe that there is utility in the approach. Making a small set of changes should virtually turn any performant SQL database into a performant execution engine for linear algebra. If one desires higher-level programming abstractions, it would be possible to implement a math-like domain specific language (such as MATLAB or SystemML's Python-like language) or API (such as a TensorFlow-like Python binding [3]) *on top* of our proposed extensions. That domain specific language or API could itself exploit high-level linear algebra transformations, and translate the computation to a database computation—with the key benefit provided by a relational backend, there is no need to implement a distributed, linear algebra execution engine from scratch.

**Our contributions.** We propose a very small set of changes to SQL that make it easy for a programmer to specify even complicated computations over vectors and matrices, and we implement our ideas in the context of the SimSQL parallel database system [13]. We show experimentally that the resulting system has performance that is comparable to a special-purpose array system (SciDB), a special-purpose scalable linear algebra system (SystemML), and a linear algebra library built directly on top of a dataflow platform (Spark's `mllib.linalg`). Our results prove the suitability of existing, relational systems for scalable linear algebra computations.

# 2. LA ON TOP OF RA

We now discuss how a relational database system might make an excellent platform for distributed linear algebra.

## 2.1 Linear and Relational Algebra

Development of distributed algorithms for linear algebra has been an active area of scientific investigation for decades, and many algorithms have become standard. Matrices to be manipulated in a distributed system are typically "blocked" or "chunked"; that is, they are divided into smaller matrices. Imagine that we want to multiply two large, dense matrices on a distributed cluster, to compute $\mathbf{O} \leftarrow \mathbf{L} \times \mathbf{R}$. We assume that the blocks of $\mathbf{L}$ are randomly located around the cluster, while the blocks from $\mathbf{R}$ are round-robin partitioned, based upon each block's row identifier.

As a first step to perform this distributed multiplication, we would shuffle the blocks from $\mathbf{L}$ so that all of the blocks from $\mathbf{L}$, column $i$ are co-located with all of the blocks from $\mathbf{R}$, row $i$. Then, at each node, a local join (in this case, a cross product) is performed to iterate through all $(\mathrm{L}j.i,\ \mathrm{R}i.k)$ pairs that can be formed at the node. For each pair, a matrix multiply is performed, so that $\mathrm{I}i.j.k \leftarrow \mathrm{L}j.i \times \mathrm{R}i.k$. Finally, all of the $\mathrm{I}i.j.k$ blocks are again shuffled so that they are co-located based upon their $(j, k)$ values—these blocks are then summed, so that the output block is computed as $\mathrm{O}j.k \leftarrow \sum_i \mathrm{I}i.j.k$.

Note that *this is really just a relational algebra computation* over the blocks making up $\mathbf{L}$ and $\mathbf{R}$. The first two steps of the computation are a distributed join that computes all $(\mathrm{L}j.i,\ \mathrm{R}i.k)$ pairs, followed by a projection that performs the matrix multiply. The next two steps—the shuffle and summation—are nothing more than a distributed grouping with aggregation.

The matrix multiplication example shows that distributed linear algebra computations are often nothing more than distributed relational algebra computations. This fact underlies our assertion that a relational database system makes an excellent platform for distributed linear algebra. Database researchers have spent decades studying efficient algorithms for distributed joins and aggregations, and relational systems are mature and performant. Using a distributed database means that there is no need to reinvent the wheel.

A further benefit of using a distributed database system as a linear algebra engine is that decades of work in query optimization is directly applicable. In our example, we decided to shuffle $\mathbf{L}$ because $\mathbf{R}$ was already partitioned on the join key. Had $\mathbf{L}$ been pre-partitioned and not $\mathbf{R}$, it would have been better to shuffle $\mathbf{R}$. This is *exactly* the sort of decision that a modern query optimizer makes with total transparency. Using a database as the basis for a linear algebra engine gives us the benefit of query optimization for free.

## 2.2 The Challenges

However, there are two main concerns associated with implementing linear algebra directly on top of an existing relational system, without modification. First is the complexity of writing linear algebra computations on top of SQL. Consider a data set consisting of the vectors $\{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_n\}$, and imagine that our goal is to compute the distance

$$d^2_{\mathbf{A}}(\mathbf{x}_i, \mathbf{x}') = (\mathbf{x}_i - \mathbf{x}')^T \mathbf{A} (\mathbf{x}_i - \mathbf{x}')$$

for a Riemannian metric [19] encoded by the matrix $\mathbf{A}$. We might wish to compute this distance between a particular data point $\mathbf{x}_i$ and every other point $\mathbf{x}'$. This would be required, for example, in a $k$NN-based classification in the metric space defined by $\mathbf{A}$.

This can be implemented in SQL as follows. Assume the set of vectors is encoded as a table:

```
data (pointID, dimID, value)
```

with the matrix $\mathbf{A}$ encoded as another table:

```
matrixA (rowID, colID, value)
```

Then, the desired computation is expressed in SQL as:

```
CREATE VIEW xDiff (pointID, dimID, value) AS
 SELECT x2.pointID, x2.dimID, x1.value - x2.value
 FROM data AS x1, data AS x2
 WHERE x1.pointID = i and x1.dimID = x2.dimID

SELECT x.pointID, SUM (firstPart.value * x.value)
FROM (SELECT x.pointID AS pointID, a.colID AS
        colID, SUM (a.value * x.value) AS value
      FROM xDiff AS x, matrixA AS a
      WHERE x.dimID = a.rowID
      GROUP BY x.pointID, a.colID)
        AS firstPart, xDiff AS x
WHERE firstPart.colID = x.dimID
      AND firstPart.pointID = x.pointID
GROUP BY x.pointID
```

While it is clearly possible to write such a code, it is not necessarily a good idea. The first problem is that this is a very intricate specification, requiring a nested subquery and a view—without the view it is even more intricate—and it bears little resemblance to the original, simple mathematics.

The second problem is perhaps less obvious from looking at the code, but just as severe: performance. This code is likely to be inefficient to execute, requiring three or four joins and two groupings. Even more concerning in practice is the fact that if the data are dense and the number of data dimensions is large (that is, there are a lot of `dimID` values for each `pointID`), then the execution of this

query will move a huge number of small tuples through the system, since a million, thousand-dimensional vectors are encoded as a billion tuples. In the classical, iterator-based execution model, there is a fixed cost incurred per tuple, which will translate to a very high execution cost. Vector-based processing can alleviate this somewhat, but the fact remains that satisfactory performance is unlikely. This fixed-cost-per-tuple problem was often cited as the impetus for designing new systems, specifically for vector- and matrix-based processing, or for processing of more general-purpose arrays.

## 2.3 The Solution

As a solution, we propose a very small set of changes to a typical relational database system that include adding new LABELED_SCALAR, VECTOR, and MATRIX data types to the relational model. Because these non-normalized data types cause the contents of vectors and matrices to be manipulated as a single unit during query processing, the simple act of adding these new types brings significant performance improvements. It becomes easy to implement linear algebra computations on top of a database with these changes.

Further, we propose a very small number of SQL language extensions for manipulating these data types and moving between them. This alleviates the complicated-code problem. In our Riemannian metric example, the two input tables data and matrixA become data (pointID, val) and matrixA (val) respectively, where data.val is a vector, and matrixA.val is a matrix. The SQL code to compute the pairwise distances becomes:

```
SELECT x2.pointID,
   inner_product (
      matrix_vector_multiply (
         a.val, x1.val - x2.val),
         x1.val - x2.val) AS value
FROM data AS x1, data AS x2, matrixA AS a
WHERE x1.pointID = i
```

# 3. OVERVIEW OF EXTENSIONS

## 3.1 New Types

At the very highest level, we propose adding VECTOR, MATRIX, and LABELED_SCALAR column types to SQL and the relational model, as well as a useful set of operations over those types (diag to extract the diagonal of a matrix, matrix_vector_multiply to multiply a matrix and a vector, matrix_multiply to multiply two matrices, and so on). Overall, 22 different built-in functions over LABELED_SCALAR, VECTOR and MATRIX types are present in our implementation. Each element of a VECTOR or a MATRIX is a double.

For a simple example of the use of VECTOR and MATRIX types, consider the following table:

```
CREATE TABLE m (mat MATRIX[10][10],
                vec VECTOR[100]);
```

This code specifies a relational table, where each tuple in the table has two attributes, mat and vec, of types MATRIX and VECTOR respectively. In our language extensions, VECTORs and MATRIXes (as above) can have specified sizes, in which case operations such as matrix_vector_multiply are automatically type-checked for size mismatches. For example, the following query:

```
SELECT matrix_vector_multiply (m.mat, m.vec)
       AS res
FROM m
```

will not compile because the number of columns in m.mat does not match the number of entries in m.vec. However, if the original table declaration had been:

```
CREATE TABLE m (mat MATRIX[10][10],
                vec VECTOR[10]);
```

then the aforementioned SQL query would compile and execute, and the output would be a database table with a single attribute (called res) of type VECTOR[10].

Note that in our extensions, there is no distinction between row and column vectors; whether or not a vector is a row or a column vector is up to the interpretation of each individual operation. matrix_vector_multiply interprets a vector as being a column vector, for example. To perform a matrix-vector multiplication treating the vector as a row vector, a programmer would first transform the vector into a one-row matrix (this transformation is described in the subsequent subsection) and then call matrix_multiply. Or, a programmer could transform the matrix first, then apply the matrix_vector_multiply function.

It is possible to create MATRIX and VECTOR types where the sizes are unspecified:

```
CREATE TABLE m (mat MATRIX[10][10],
                vec VECTOR[]);
```

In this case, the aforementioned matrix_vector_multiply SQL query would compile, but there could possibly be a runtime error if one or more of the tuples in m contained a vec attribute that did not have 10 entries.

It is also possible to have a MATRIX declaration where only one of the dimensionalities is given; for example, MATRIX[10][] is acceptable. However, if dimensions are known, it can help the optimization process because the optimizer is aware of the sizes of intermediate results.

## 3.2 Built-In Operations

In addition to a long list of standard linear algebra operations, the standard arithmetic operations +, −, ⋆ and / (element-wise) are also defined over MATRIX and VECTOR types. For example:

```
CREATE TABLE m (mat MATRIX[100][10]);

SELECT mat * mat
FROM m
```

returns a database table which stores the Hadamard product of each matrix in m with itself.

Since the standard arithmetic operations are all overloaded to work with MATRIX and VECTOR types, it means that the standard SQL aggregate operations all work as expected automatically. The SUM aggregate over MATRIX type attribute, for example, performs a + (entry-by-entry addition) over each MATRIX in a relation. This can be very convenient for implementing mathematical computations. For example, imagine that we have a matrix stored as a relational table of vectors, and we wish to perform a standard Gram matrix computation (if the matrix $\mathbf{X}$ is stored as a set of columns $\mathbf{X} = \{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_n\}$, then the gram matrix of $\mathbf{X}$ is $\sum_{i=1}^{n} \mathbf{x}_i \mathbf{x}_i^T$). This computation can be implemented simply as:

```
CREATE TABLE v (vec VECTOR[]);

SELECT SUM (outer_product (vec, vec))
FROM v
```

Arithmetic between a scalar value and a MATRIX or VECTOR type performs the arithmetic operation between the scalar and *every* entry in the MATRIX or VECTOR. In this way, it becomes very easy to specify linear algebra computations of significant complexity using just a few lines of code. For example, consider the problem of learning a linear regression model. Given a matrix $\mathbf{X} = \{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_n\}$ and a set of outcomes $\{y_1, y_2, ..., y_n\}$, the goal is to estimate a vector $\hat{\boldsymbol{\beta}}$ where for each $i$, $\mathbf{x}_i \hat{\boldsymbol{\beta}} \approx y_i$. In prac-

tice, $\hat{\boldsymbol{\beta}}$ is typically computed so as to minimize the squared loss $\sum_i (\mathbf{x}_i \hat{\boldsymbol{\beta}} - y_i)^2$. In this case, the formula for $\hat{\boldsymbol{\beta}}$ is given as:

$$\hat{\boldsymbol{\beta}} = \left( \sum_i \mathbf{x}_i \mathbf{x}_i^T \right)^{-1} \left( \sum_i \mathbf{x}_i y_i \right)$$

This can be coded as follows. If we have:

```
CREATE TABLE X (i INTEGER, x_i VECTOR []);
CREATE TABLE y (i INTEGER, y_i DOUBLE);
```

then the SQL code to compute $\hat{\boldsymbol{\beta}}$ is:

```
SELECT matrix_vector_multiply (
    matrix_inverse (
        SUM (outer_product (X.x_i, X.x_i))),
        SUM (X.x_i * y_i))
FROM X, y
WHERE X.i = y.i
```

Note the multiplication `X.x_i * y_i` between the vector `X.x_i` and the scalar `y_i`, which multiplies `y_i` by each entry in `X.x_i`.

## 3.3 Moving Between Types

By introducing `MATRIX` and `VECTOR` types, we then have new, de-normalized alternatives for storing data. For example, a matrix can be stored as a traditional triple-entry relation:

```
mat (row INTEGER, col INTEGER, value DOUBLE)
```

or as a relation containing a set of row vectors, or as a set of column vectors using

```
row_mat (row INTEGER, vec_value VECTOR[])
```

or

```
col_mat (col INTEGER, vec_value VECTOR[])
```

Or, the matrix can be stored as a relation with a single tuple having the whole matrix:

```
mat (value MATRIX [][])
```

It is of fundamental importance to be able to move around between these various representations, for several reasons. Most importantly, each has its own performance characteristics and ease-of-use for various tasks; depending upon a particular computation, one may be preferred over another.

Reconsider the linear regression example. Had we stored the data as:

```
CREATE TABLE X (mat MATRIX [][]);
CREATE TABLE y (vec VECTOR []);
```

then the SQL code to compute $\hat{\boldsymbol{\beta}}$ would have been:

```
SELECT matrix_vector_multiply (
    matrix_inverse (
      matrix_multiply (trans_matrix (mat), mat)),
    matrix_vector_multiply (
      trans_matrix (mat), vec))
FROM X, y
```

Arguably, this is a more straightforward translation of the mathematics compared to the code that stores `X` as a set of vectors. However, it may not perform as well because it may be more difficult to parallelize on a shared-nothing cluster of machines. In comparison to the vector-based implementation, the matrix multiply $\mathbf{X}^T\mathbf{X}$ is implicit in the relational algebra.

Since different representations are going to have their own merits, it may be necessary to construct (or deconstruct) `MATRIX` and `VECTOR` types using SQL. To facilitate this, we introduce the notion of a *label*. In our extension, each `VECTOR` attribute implicitly or explicitly has an integer label value attached to it (if the label is never explicitly set for a particular vector, then its value is $-1$ by default). In addition, we introduce a new type called `LABELED_SCALAR`, which is essentially a `DOUBLE` with a label. Using those labels along with three special aggregate functions (`ROWMATRIX`, `COLMATRIX`, and `VECTORIZE`), it is possible to write SQL code that creates `MATRIX` types and `VECTOR` types, respectively, from normalized data.

For example, reconsider the table:

```
CREATE TABLE y (i INTEGER, y_i DOUBLE);
```

Imagine that we want to create a table with a single vector tuple from the table `y`. To do this, we simply write:

```
SELECT VECTORIZE (label_scalar (y_i, i))
FROM y
```

Here, the `label_scalar` function creates an attribute of type `LABELED_SCALAR`, attaching the label `i` to the `DOUBLE` `y_i`. Then, the `VECTORIZE` operation aggregates the resulting values into a vector, adding each `LABELED_SCALAR` value to the vector at the position indicated by the label. Any "holes" (or entries in the vector for which no `LABELED_SCALAR` were found) in the resulting vector are set to zero. The number of entries in the vector is set to be equal to the largest label of any entry in the vector.

As stated above, `VECTOR` attributes implicitly have labels, but they can be set explicitly as well, and those labels can be used to construct matrices. For example, imagine that we want to create a single tuple with a single matrix from the table:

```
mat (row INTEGER, col INTEGER, value DOUBLE)
```

We can do this with the following SQL code:

```
CREATE VIEW vecs AS
 SELECT VECTORIZE (label_scalar (val, col))
     AS vec, row
 FROM mat
 GROUP BY row
```

followed by:

```
SELECT ROWMATRIX (label_vector (vec, row))
FROM vecs
```

The first bit of code creates one vector for each row, and the second bit of code aggregates those vectors into a matrix, using each vector as a row. It would have been possible to create a column matrix by first using a `GROUP BY col` and then `SELECT COLMATRIX`.

So far we have discussed how to de-normalize relations into vectors and matrices. It is equally easy to normalize `MATRIX` and `VECTOR` types. Assuming the existence of a table `label (id)` which simply lists the values 1, 2, 3, and so on, then one can move from the vectorized representation to a purely-relational representation using a `join` of the form:

```
SELECT label.id, get_scalar (vecs.vec, label.id)
FROM vecs, label
```

Code to normalize a matrix is written similarly.

## 3.4 Local Matrix vs. Distributed Matrix

In keeping with a traditional RDBMS design, our system enforces that all vectors and matrices should be small enough to fit into the RAM of an individual machine. Since our mantra is "incremental, not revolutionary," and distributing individual tuples or attributes across machines is generally not supported by modern database systems, it seems reasonable not to support distributed vector/matrix data types in our system.

Of course, one might ask, *What if one has a matrix that is too large to fit into the RAM of an individual machine?* Fortunately, it turns out that our extension can handle this easily and efficiently.

For example, a large, dense matrix with 100,000 rows and 100,000 columns can be stored as one hundred tuples in the table:

```
bigMatrix (tileRow INTEGER, tileCol INTEGER,
    mat MATRIX[10000][10000])
```

Efficient, distributed matrix operations are then easily possible via SQL. For example, to multiply `bigMatrix` with `anotherBigMat (tileRow, tileCol, mat)`, we would use:

```
SELECT lhs.tileRow, rhs.tileCol,
  SUM (matrix_multiply (lhs.mat, rhs.mat))
FROM bigMatrix AS lhs, anotherBigMat AS rhs
WHERE lhs.tileCol = rhs.tileRow
GROUP BY lhs.tileRow, rhs.tileCol
```

# 4. TYPING AND OPTIMIZATION

## 4.1 Vector and Matrix Sizes

In practical applications, the individual matrices stored in a database table can range from a few bytes in size to many gigabytes in size. Hence, knowing the size of an individual linear algebra object stored in a database is going to be of fundamental importance during query optimization. Unfortunately, linear algebra objects are typically manipulated via a large set of user-defined and system-provided functions that change the sizes of the objects being manipulated in ways that are regular, but opaque to the system. This can easily result in the choice of a query plan that is far from optimal.

The problem can be illustrated by a simple example. Assume we have three tables defined as below:

```
R (r_rid INTEGER, r_matrix MATRIX[10][100000])
S (s_sid INTEGER, s_matrix MATRIX[100000][100])
T (t_rid INTEGER, t_sid INTEGER)
```

Imagine that the sizes of the tables `R`, `S`, and `T` are 100 tuples, 100 tuples, and 1,000 tuples, respectively. Now, suppose we want to calculate the product of a number of pairs of matrices from the relations `R` and `S`, where the pairs for which we need to obtain are indicated by `T`:

```
SELECT matrix_multiply (r_matrix, s_matrix)
FROM R, S, T
WHERE r_rid = t_rid AND s_sid = t_sid
```

A rule-based optimizer, or a cost-based optimizer without access to good information about the size of the linear algebra object being pushed through the system is almost assuredly going to choose a plan such as $\pi((\texttt{S} \bowtie \texttt{T}) \bowtie \texttt{R})$ where the projection $\pi$ contains the matrix multiply. It will not join `R` and `S` first because no join predicate links them. In this plan, the join between tables `S` and `T` produces about 1,000 tuples (estimated as $\frac{1000 \times 100}{100}$), each containing an 80MB matrix (estimated as $8 \times 100000 \times 100$ bytes). Thus, the total data produced in this join is about 80 GB.

However, this is clearly not the optimal query plan. It is possible to do a lot better using the plan $(\pi(\texttt{S} \times \texttt{R})) \bowtie \texttt{T}$, where the projection $\pi$ again contains the matrix multiply. While the cross product between the tables `S` and `R` produces 10,000 tuples, the early projection allows the optimizer to produce a plan that performs the `matrix_multiply (r_matrix, s_matrix)` early, to effectively remove all of the large matrices from the plan; the result of each matrix multiply is only 8KB (estimated as $8 \times 10 \times 100$ bytes). Thus, the total data produced in this join and projection is about 80 MB, and it is likely far superior.

## 4.2 Type Signatures

To make sure that the database optimizer has the information necessary to choose the correct plan, the type signature for any function that includes vectors and matrices is *templated*. The type signature takes (as an argument) the size and shape of the input, and returns the size and shape of the output. For example, the function signature of the built-in function `diag` (computing the diagonal of a matrix) is:

```
diag(MATRIX[a][a]) -> VECTOR[a]
```

This signature constrains the input matrix to be square, and it indicates that the output vector has a number of entries identical to the number of rows/columns of the input matrix. The signature for `matrix_multiply` is:

```
matrix_multiply(MATRIX[a][b], MATRIX[b][c]) ->
          MATRIX[a][c]
```

In this signature, the arguments `a`, `b`, and `c` effectively parameterize the function signature. This information is then used by the optimizer to infer the exact dimensions of the output object. For example, consider the schema:

```
U (u_matrix MATRIX[1000][100])
V (v_matrix MATRIX[100][10000])
```

And the query:

```
SELECT matrix_multiply(u_matrix, v_matrix)
FROM U, V
```

The optimizer obtains the dimensions of the `u_matrix` and `v_matrix` objects by looking in the catalog. When the dimensions of `u_matrix` are retrieved from the catalog, the type parameter `a` is bound to `1000`, and `b` is bound to `100`. When the dimensions of `v_matrix` are retrieved, `b` is bound a second time to `100` (a different value for `b` would cause a compile-time error) and `c` is bound to `10000`. Hence, the output of the matrix multiply is a 1000-by-10000 matrix of approximately 80 MB in size; this information can subsequently be used by the optimizer.

# 5. EXPERIMENTS

We have implemented all of the capabilities described in the paper on top of SimSQL [13], a prototype Java- and Hadoop-based database designed for scalable analytics. In this section, we experimentally evaluate the utility of the new capabilities by comparing SimSQL to a number of alternative platforms.

**Platforms Tested.** The platforms we evaluated are:

(1) SimSQL. We tested several different SimSQL implementations: Without vector/matrix support (the original SimSQL implementation, without the improvements proposed in this paper), with data stored as vectors, and with data stored as vectors, then converted into blocks.

(2) SystemML. This is SystemML V0.9, which provides the option to run on top of Hadoop. All computations are written in SystemML's DML programming language.

(3) SciDB. This is SciDB V14.8. All computations are written in SciDB's AQL language which is similar to SQL.

(4) Spark `mllib.linalg`. This is run on Spark V1.6 in standalone mode. All computations are written in Scala.

**Computations Performed.** In our experiments, we performed three different representative computations.

(1) Gram matrix computation. A Gram matrix is the inner products of a set of vectors. It is a common computational pattern in machine learning, and is often used to compute the kernel functions and covariance matrices. If we use a matrix $\mathbf{X}$ to store the input vectors, then the Gram matrix $\mathbf{G}$ can be calculated as $\mathbf{G} = \mathbf{X}^T\mathbf{X}$.

(2) Least squares linear regression. Given a paired data set $\{y_i, \mathbf{x}_i\}$, $i = 1, \ldots, n$, we wish to model each $y_i$ as a linear combination of the values in $\mathbf{x}_i$. Let $y_i \approx \mathbf{x}_i^T \boldsymbol{\beta} + \epsilon_i$, where $\boldsymbol{\beta}$ is the vector of regression coefficients. The most common estimator for $\boldsymbol{\beta}$ is the least squares estimator: $\hat{\boldsymbol{\beta}} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$.

(3) Distance computation. We first compute the distance between each data point pair $\mathbf{x}_i$ and $\mathbf{x}'$: $d^2_{\mathbf{A}}(\mathbf{x}_i, \mathbf{x}') = \mathbf{x}_i^T\mathbf{A}\mathbf{x}'$. Then, for each data point $\mathbf{x}_i$, we compute the minimum $d^2_{\mathbf{A}}(\mathbf{x}_i, \mathbf{x}')$ value over all $\mathbf{x}' \neq \mathbf{x}_i$. Lastly, we select the data points which have the max value among those minimums.

**Implementation Details.** We now describe in some detail how we performed each of these three computations over the various platforms.

(1) SimSQL. A SimSQL programmer uses queries and built-in functions to conduct computations. In SimSQL, we implemented each model using three different SQL codes. First, we wrote a pure-tuple based code (as on an existing, standard SQL-based platform). Second, we wrote an SQL code where each data point is stored as an individual vector, in the schema:

```
x_vm (id INTEGER, value VECTOR[])
```

Third, we wrote an SQL code where data points are grouped together in blocks of 1000 data points, and stored as a matrix with 1000 rows, so that they can be manipulated as a group.

The Gram matrix computation is written over tuples as:

```
SELECT x1.col_index, x2.col_index,
       SUM(x1.value * x2.value)
FROM x AS x1, x AS x2
WHERE x1.row_index = x2.row_index
GROUP BY x1.col_index, x2.col_index;
```

The Gram matrix is computed over vectors as:

```
SELECT SUM(outer_product(x.value, x.value))
FROM x_vm AS x;
```

For a block-based computation, the rows are first grouped into blocks (the table `block_index (mi INTEGER)` stores the indices for blocks):

```
CREATE VIEW MLX (m) AS
  SELECT ROWMATRIX(label_vector(
       x.value, x.id - ind.mi*1000))
  FROM x_vm AS x, block_index AS ind
  WHERE x.id/1000 = ind.mi
  GROUP BY ind.mi;
```

Note that this grouping step is not necessary if the data are already stored as blocks; in our experiments, we count the blocking time as part of the computation.

Then, the result is a sum of a series of matrix multiplies:

```
SELECT SUM(matrix_multiply(
       trans_matrix(mlx.m), mlx.m))
FROM mlx;
```

The calculation of linear regression is similar to Gram matrix computation. We omit the code for brevity. We also omit the code for tuple-based distance computation.

The key codes of vector-based and block-based distance computation are given below. For the vector-based computation, we calculate the minimum $d^2_{\mathbf{A}}(\mathbf{x}_i, \mathbf{x}')$ for each data point $\mathbf{x}_i$ as (the table `MX` stores the distances computed by another query):

```
CREATE VIEW DISTANCESM (id, dist) AS
   SELECT a.dataID,
        MIN (inner_product (mxx.mx_data, a.data))
   FROM X_m AS a, MX AS mxx
   WHERE a.dataID <> mxx.id
   GROUP BY a.dataID;
```

And in the block-based computation, we first conduct the computation $\mathbf{x}_i^T\mathbf{A}\mathbf{x}'$ via a set of matrix multiplies:

```
CREATE VIEW DISTANCES (id1, id2, dm) AS
   SELECT mxx.id, mx.id, matrix_multiply(
     mxx.m, matrix_multiply(mp.mapping,
     trans_matrix(mx.m)))
   FROM MLX AS mx, MLX AS mxx, MM AS mp;
```

Then, the minimum values of those computations for each data point is calculated via a series of operations on matrices.

(2) SystemML. Physically, the data in SystemML are stored and processed as *blocks*, which are square matrices.

Gram matrix computation in SystemML is:

```
result = t(X) %*% X
```

Linear regression is omitted. The code of distance computation is:

```
all_dist = X %*% m %*% X_t
all_dist = all_dist + diag(diag_inf)
min_dist = rowMins(all_dist)
result = rowIndexMax(t(min_dist))
```

(3) Spark `mllib.linalg`. A Spark `mllib.linalg` programmer must decide: should the input data be stored/processed as vectors, or as matrices? And, if a matrix is used, should it be a local matrix, or a distributed one? In our experiments, we tried different vector/local matrix/distributed matrix implementations, and selected the most efficient ones.

For Gram matrix computation, vector-based is the fastest:

```
val result = parsedData.map(
      x => x.transpose.multiply(
                x.asInstanceOf[DenseMatrix]
                ).toArray
      ).reduce((a, b) => (a, b).zipped.map(_+_))
```

For linear regression, vector-based is also the most efficient. We omit the code for brevity.

The distance computation was challenging. After a lot of experimentation, we found that the distributed `BlockMatrix` was the best. The code is as follows:

```
val dist_matrix = block_matrix_x.
        multiply(block_matrix_m).
        multiply(block_matrix_x.transpose)

val result =
    dist_matrix.toIndexedRowMatrix.rows.map(
    x => (x.index, x.vector.toArray)).
     map{ case(i, a) =>
          {if (i==0) a(0)=a(1)
             else a(i.toInt)=a(0); (i, a.min);}
    }.max()(
        new Ordering[Tuple2[Long, Double]]() {
          override def compare(
            x: (Long, Double), y: (Long, Double)
          ): Int =
          Ordering[Double].compare(x._2, y._2)})
```

(4) SciDB. Data in SciDB are partitioned as *chunks*. We use 1000 as the chunk size for all arrays in our code.

The SciDB code of Gram matrix computation is:

```
SELECT * FROM gemm(transpose(x), x,
                build(<val:double>[t1=0:9,1000,0,
                    t2=0:9,1000,0], 0));
```

Linear regression is similar. The implementation of the distance computation is:

| Gram Matrix Computation | | | |
|---|---|---|---|
| Platform | 10 dims | 100 dims | 1000 dims |
| Tuple SimSQL | 00:01:28 | 00:03:19 | 05:04:45 |
| Vector SimSQL | 00:00:37 | 00:00:43 | 00:05:43 |
| Block SimSQL | 00:01:18 | 00:01:23 | 00:02:53 |
| SystemML | 00:00:05∗ | 00:00:51 | 00:02:34 |
| Spark `mllib` | 00:00:20 | 00:00:54 | 00:17:31 |
| SciDB | 00:00:03 | 00:00:17 | 00:03:20 |

Figure 1: Gram matrix results. Format is HH:MM:SS. A star (∗) indicates running in local mode.

```
SELECT * INTO mxt
FROM gemm(m, transpose(x),
        build(<val:double>[t1=0:999,1000,0,
                t2=0:99999,1000,0], 0));

SELECT * INTO all_distance
FROM filter(gemm(x, mxt,
      build(<val:double>[t1=0:99999,1000,0,
        t2=0:99999,1000,0], 0)), t1<>t2);

SELECT min(gemm) INTO distance
FROM all_distance
GROUP BY t1;

SELECT * INTO max_dist
FROM (SELECT max(min) FROM distance);

SELECT t1
FROM distance JOIN max_dist ON
        distance.min = max_dist.max;
```

**Experiment Setup.** We ran all experiments on 10 Amazon EC2 m2.4xlarge machines (as workers), each having eight CPU cores. For Gram matrix computation and linear regression, the number of data points per machine was $10^5$. For the distance computation, the number of data points per machine was $10^4$. All data sets were dense, and all data were synthetic—since we are only interested in running time; there is likely no practical difference between synthetic and real data. For each computational task, we considered three data dimensionalities: 10, 100, and 1000.

**Experiment Results and Discussion.** The results are shown in Figures 1, 2, and 3.

Vector- and block-based SimSQL clearly dominate the tuple-based implementation for each of the three computations. To examine this further, we re-ran the tuple-based and vector-based Gram matrix computations over 1000-dimensional data on a five machine cluster, and timed the individual operations that made up the computation (shown in Figure 4). Note that in the 1000-dimensional computation, in the tuple-based computation, each tuple joins with the other 1000 values making up the same data point, and all of those tuples need to be aggregated. Since $5 \times 10^5$ data points are stored as $5 \times 10^8$ tuples, this results in $5 \times 10^{11}$ tuples that need to be aggregated. Even though these operations are pipelined, they dominate the running time, as shown in Figure 4. Here we see—perhaps surprisingly—that the the dominant cost is *not* the `join` in the tuple-based computation, but the `aggregation`. This illustrates the problem with tuple-based linear algebra: even a tiny fixed cost associated with each tuple is magnified when we must push $5 \times 10^{11}$ tuples through the system.

Interestingly, we see that the vector-based computation was faster than block-based for 10- and 100-dimensional computations. This is because our experiments counted the time of grouping vectors into blocked matrices. This additional computation was not worthwhile for less computationally expensive problems. But for the 1000-dimensional computations, additional time savings could be realized via blocking.

| Linear Regression | | | |
|---|---|---|---|
| Platform | 10 dims | 100 dims | 1000 dims |
| Tuple SimSQL | 00:03:42 | 00:05:46 | 05:05:22 |
| Vector SimSQL | 00:00:45 | 00:00:49 | 00:06:35 |
| Block SimSQL | 00:02:23 | 00:02:22 | 00:04:22 |
| SystemML | 00:00:06∗ | 00:00:53 | 00:02:38 |
| Spark `mllib` | 00:00:35 | 00:01:01 | 00:17:42 |
| SciDB | 00:00:15 | 00:00:33 | 00:06:04 |

Figure 2: Linear regression results. Format is HH:MM:SS. A star (∗) indicates running in local mode.

| Distance Computation | | | |
|---|---|---|---|
| Platform | 10 dims | 100 dims | 1000 dims |
| Tuple SimSQL | Fail | Fail | Fail |
| Vector SimSQL | 00:10:14 | 00:11:49 | 00:13:53 |
| Block SimSQL | 00:03:14 | 00:04:43 | 00:10:36 |
| SystemML | 00:13:29 | 00:22:38 | 00:33:22 |
| Spark `mllib` | 01:22:59 | 01:15:06 | 01:13:06 |
| SciDB | 00:03:46 | 00:04:54 | 00:05:06 |

Figure 3: Distance computation results. Format is HH:MM:SS.

For the higher-dimensional computations, there was no clear winner among SystemML, SciDB, and SimSQL. SimSQL was a bit slower for the lower-dimensional problems, because, as a prototype system, it is not engineered for high throughput. Spark `mllib` was not competitive on the higher-dimensional data. Over the three, 1000-dimensional computations, SimSQL, SystemML, and SciDB had geometric mean running times of 5 minutes 7 seconds, 6 minutes 5 seconds, and 4 minutes 41 seconds, respectively.

We spent a lot of time trying to tune both SimSQL and SystemML for the distance computation. In the case of SimSQL, the problem appears to be that there are only $10^5$ data points in all; when grouped into blocks of 1000 vectors, this results in only 100 matrices in all. This meant that each of our 80 compute cores had an average of 1.25 matrices mapped to it. Since SimSQL uses a randomized, hash-based partitioning, it is easily possible for one core to receive four or five of the 100 matrices. We did observe that most cores would finish in a short time, while just a few, overloaded cores would be left to finish the computation in a much longer period. Better load balancing would likely have solved this problem.

Finally, we ask the question: do these experiments support the hypothesis at the core of the paper, that a relational engine can be used with little modification to support efficient linear algebra processing? In terms of performance, they seem to. Enhancing a relatively slow, Java-based system (SimSQL) resulted in a relational algebra system with very reasonable performance for linear algebra computations.

# 6. RELATED WORK

There has been some recent interest in combining distributed/-parallel data management systems and linear algebra to support analytics. One approach is the construction of a special purpose data management system for scalable linear algebra; SystemML [16] is the best example of this. Another good example of this is the Cumulon system [18], which has the notable capability of optimizing its own hardware settings in the cloud. MadLINQ [21], built on top of Microsoft's LINQ framework, can also be seen as an example of this. Other work aims at scaling statistical/numerical programming languages such as R. Ricardo [15] aims to support R programming on top of Hadoop. Riot [25] attempts to plug an I/O efficient back-

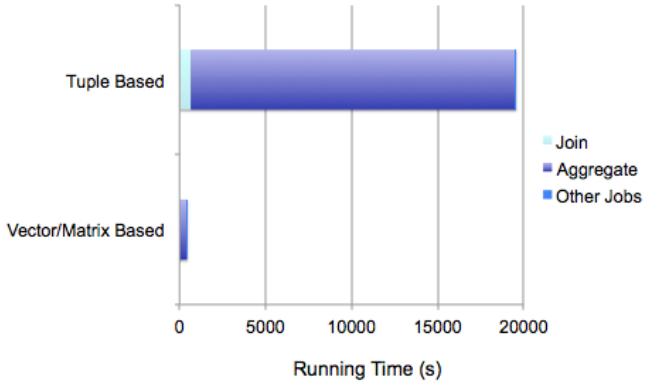


Figure 4: Comparison of Gram matrix computation for tuple-based and vector-based SimSQL.

end into R to bring scalability.

A second (and not completely distinct) approach is building scalable linear algebra libraries on top of a dataflow platform. In this paper, we have experimentally considered `mllib.linalg` [1]. Apache Hama [22] is another example of such a package. So is SciHadoop [12].

The idea of moving past relations onto arrays as a database data model, particularly for scientific and/or numerical applications, has been around for a long time. One of the most notable efforts is Baumann and his colleague's work on Rasdaman [6]. In this paper, we have compared with SciDB [11], an array database for which linear algebra is a primary use case.

An array-based approach that is somewhat related to what we have proposed is SciQL [24], which is a system supporting an extended SQL that is implemented on top of the MonetDB system [10]. SciQL adds arrays (in addition to tables) as a second data storage abstraction. Our proposed approach is much more modest; rather than allowing arrays as a fundamental data abstraction, we simply add vectors and matrices as new attribute types.

There is some support for linear algebra in modern, commercial relational database systems, but it is not well-integrated into the declarative (`SELECT-FROM-WHERE`) portion of SQL, and generally challenging to use. For example, Oracle provides the UTL_NLA [2] package to support BLAS and LAPACK operations. To multiply two matrices using this package, and assuming two input matrices `m1` and `m2` declared as type `utl_nla_array_dbl` (and an output matrix `res` defined similarly), a programmer would write:

```
utl_nla.blas_gemm(
 transa => 'N', transb => 'N', m => 3, n => 3,
 k => 3, alpha => 1.0, a => m1, lda => 3,
 b => m2, ldb => 2, beta => 0.0, c => res,
 ldc => 3, pack => R);
```

There have been efforts [17, 20] aimed at building analytics libraries, including linear algebra functionality, on top of a database system. However, these efforts use (external) tools such as user defined functions to build linear algebra on top of a database system.

# 7. CONCLUSIONS

We have proposed a small set of changes to SQL that can render any distributed, relational database engine a high-performance platform for distributed linear algebra. We have shown that making these changes to a distributed relational database (SimSQL) results in a system for distributed linear algebra whose performance meets or exceeds special-purpose systems. Given that SimSQL is a prototype system written mostly in Java, it is not unreasonable to speculate that a commercial, high-performance database system with similar extensions could do even better. We believe that our results call into question the need to build yet another special-purpose data management system for linear-algebra-based analytics.

**Acknowledgments.** Material in this paper has been supported by the NSF under grant nos. 1355998 and 1409543, and by the DARPA MUSE program.

# 8. REFERENCES

[1] Apache spark mllib: `http://spark.apache.org/docs/latest/mllib-data-types.html`.

[2] Oracle corporation: `https://docs.oracle.com/cd/B19306_01/index.htm`.

[3] M. Abadi et al. Tensorflow: Large-scale machine learning on heterogeneous systems, 2015. *Software available from tensorflow.org*.

[4] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' guide*, volume 9. Siam, 1999.

[5] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, et al. Spark sql: Relational data processing in spark. In *SIGMOD*, pages 1383–1394. ACM, 2015.

[6] P. Baumann, A. Dehmel, P. Furtado, R. Ritsch, and N. Widmann. The multidimensional database system rasdaman. In *SIGMOD Record*, volume 27, pages 575–577. ACM, 1998.

[7] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, et al. *ScaLAPACK users' guide*, volume 4. siam, 1997.

[8] L. S. Blackford, A. Petitet, R. Pozo, K. Remington, R. C. Whaley, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, et al. An updated set of basic linear algebra subprograms (blas). *ACM TOMS*, 28(2):135–151, 2002.

[9] M. Boehm, D. R. Burdick, A. V. Evfimievski, B. Reinwald, F. R. Reiss, P. Sen, S. Tatikonda, and Y. Tian. Systemml's optimizer: Plan generation for large-scale machine learning programs. *IEEE Data Eng. Bull.*, 37(3):52–62, 2014.

[10] P. A. Boncz, M. Zukowski, and N. Nes. Monetdb/x100: Hyper-pipelining query execution. In *CIDR*, volume 5, pages 225–237, 2005.

[11] P. G. Brown. Overview of SciDB: large scale array storage, processing and analysis. In *SIGMOD*, pages 963–968, 2010.

[12] J. B. Buck, N. Watkins, J. LeFevre, K. Ioannidou, C. Maltzahn, N. Polyzotis, and S. Brandt. SciHadoop: Array-based query processing in Hadoop. In *ACM SC*, page 66, 2011.

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

[14] S. Chaudhuri. An overview of query optimization in relational systems. In *PODS*, pages 34–43. ACM, 1998.

[15] S. Das, Y. Sismanis, K. S. Beyer, R. Gemulla, P. J. Haas, and J. McPherson. Ricardo: integrating R and Hadoop. In *SIGMOD*, pages 987–998, 2010.

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

[17] J. M. Hellerstein, C. Ré, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li, et al. The MADlib analytics library: or MAD skills, the SQL. *VLDB*, 5(12):1700–1711, 2012.

[18] B. Huang, S. Babu, and J. Yang. Cumulon: Optimizing statistical data analysis in the cloud. In *SIGMOD*, pages 1–12, 2013.

[19] G. Lebanon. Metric learning for text documents. *IEEE PAMI*, 28(4):497–508, 2006.

[20] C. Ordonez. Statistical model computation with udfs. *IEEE TKDE*, 22(12):1752–1765, 2010.

[21] Z. Qian, X. Chen, N. Kang, M. Chen, Y. Yu, T. Moscibroda, and Z. Zhang. Madlinq: large-scale distributed matrix computation for the cloud. In *EuroSys*, pages 197–210. ACM, 2012.

[22] S. Seo, E. J. Yoon, J. Kim, S. Jin, J.-S. Kim, and S. Maeng. Hama: An efficient matrix computation with the mapreduce framework. In *CloudCom*, pages 721–726. IEEE, 2010.

[23] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a warehousing solution over a map-reduce framework. *VLDB*, 2(2):1626–1629, 2009.

[24] Y. Zhang, M. Kersten, and S. Manegold. Sciql: array data processing inside an rdbms. In *SIGMOD*, pages 1049–1052. ACM, 2013.

[25] Y. Zhang, W. Zhang, and J. Yang. I/o-efficient statistical computing with riot. In *ICDE*, pages 1157–1160. IEEE, 2010.

# Technical Perspective: Toward Building Entity Matching Management Systems

Wang-Chiew Tan
Recruit Institute of Technology
wangchiew@recruit.ai

*Entity matching*, also known as *entity resolution* or *reference reconciliation*, is to identify when two (different) representations refer to the same real-world entity. Overcoming the entity matching problem is often a key step in today's data preparation and integration pipeline before useful data can be produced for analysis. For example, to understand how many potential new customers there may be, a company may wish to integrate an internal repository of customer profiles to an externally sourced dataset that contains profiles of users (e.g., Twitter data). A successful entity matching process would need to discern when two heterogeneous customer profiles may actually refer to the same customer and also for the opposite, when two seemingly identical customer profiles may actually not be the same customer. For example, it is not obvious whether or not the these two records:

(`D. Smith`, `IBM Yorktown`, ...)
(`S., David`, `International Business Machine`, ...)

refer to the same person and one may need to understand the remaining values in the customer profiles before a final decision can be made. An entity matching outcome is largely dependent on the features that are selected by the user or learned (if training data is available) for determining whether a match is successful. Different features and measures may lead to different outcomes for the two records above. Similarly, a different training data used for training the entity matching model may lead to a different outcome. On top of this, for an entity matching workflow to be useful, more often than not, it has to scale to large datasets.

Magellan is a fairly recent entity matching system developed at the University of Wisconsin that overcomes several limitations of existing solutions to entity matching. There are two important attributes of Magellan that make it particularly useful and "easy" for end users to develop entity matching solutions and this paper describes how Magellan has been successfully used by several such end-user groups.

First, Magellan has a rich set of libraries for users to carry out the entire entity matching pipeline which may involve several substeps such as data cleaning, visualization, in addition to blocking, and matching. For example, Magellan provides libraries for different types of string matching functions (a basic building block in entity matching). And because Magellan is developed in Python, it can also leverage publicly available Python libaries (more than 130,000 libraries are available for data science, see pypi.python.org) for other tasks such as exploration, visualization, and cleaning.

Second, Magellan provides how-to guides that describe how to approach the development of entity matching workflows. The how-to guides are useful because they describe step-by-step instructions with examples that illustrate the example methodology and functionalities available in Magellan. In addition, they illustrate critical substeps that are sometimes overlooked by users in designing an entity matching workflow. For example, during the design of an entity matching workflow for large datasets, one often *downsamples* the dataset so that the resulting dataset is smaller and allows for faster testing. However, care has to be taken to ensure that the downsampled dataset is representative of the original dataset and Magellan provides supporting tools to help with the downsampling process.

The user works in the *development stage* with the downsampled dataset. When ready, the user moves the final entity matching workflow to the *production stage* where it will be executed on the original dataset with supporting software libraries for scaling the operation such as running MapReduce/Spark jobs in a parallel and distributed setting.

The paper largely describes the development stage by delineating a few main steps in the development of entity matching workflows that uses supervised learning. Magellan provides a tool for downsampling which samples data intelligently to ensure a reasonable number of matches exists in the downsampled dataset. After this, the downsampled data is *blocked* to remove tuples that are highly unlikely to match. Blocking helps further reduce the number of candidate matches to consider and can considerably speed up the overall entity matching process. Magellan provides a debugger tool for users to examine the tuples that are eliminated by the blocker. A general rule of thumb is that if there are only a few matches among the eliminated tuples, then the blocker has achieved a sufficiently high reall. From the remaining tuples, the next step samples a set of candidate matching tuples and the user labels the candidates as match/no-match. Magellan provides tools to ensure that there are sufficiently many true matches in the sampled data. After this, Magellan automatically generates a set of features from the labeled data and converts each candidate pair of tuples into a feature vector. By training and cross validation, it selects a matcher with the the highest estimated accuracy from among those supplied by Magellan. A debugging step then follows to examine the mistakes of the matcher and improve upon it as needed and the process can be repeated.

# Magellan: Toward Building Entity Matching Management Systems

Pradap Konda[1], Sanjib Das[1], Paul Suganthan G.C.[1], Philip Martinkus[1], AnHai Doan[1],
Adel Ardalan[1], Jeffrey R. Ballard[1], Yash Govind[1], Han Li[1],
Fatemah Panahi[2], Haojun Zhang[1], Jeff Naughton[2],
Shishir Prasad[3], Ganesh Krishnan[3], Rohit Deep[3], Vijay Raghavendra[3]
[1]University of Wisconsin-Madison, [2]Google, [3]@WalmartLabs

## ABSTRACT

Entity matching (EM) has been a long-standing challenge in data management. Most current EM works focus only on developing matching algorithms. We argue that far more efforts should be devoted to building EM systems. We discuss the limitations of current EM systems, then describe Magellan, a new kind of EM system. Magellan is novel in four important aspects. (1) It provides how-to guides that tell users what to do in each EM scenario, step by step. (2) It provides tools to help users execute these steps; the tools seek to cover the entire EM pipeline, not just blocking and matching as current EM systems do. (3) Tools are built into the Python open-source data science ecosystem, allowing Magellan to borrow a rich set of capabilities in data cleaning, IE, visualization, learning, etc. (4) Magellan provides a powerful scripting environment to facilitate interactive experimentation and quick "patching" of the system. We describe research challenges and present extensive experiments that show the promise of the Magellan approach.

## 1. INTRODUCTION

Entity matching (EM) identifies data instances that refer to the same real-world entity, such as (David Smith, UW-Madison) and (D. M. Smith, UWM). This problem has been a long-standing challenge in data management [13, 22]. Most current EM works however have focused only on developing *matching algorithms* [13, 22].

Going forward, we believe that *building EM systems* is truly critical for advancing the field. EM is engineering by nature. We cannot just keep developing matching algorithms in a vacuum. This is akin to continuing to develop ever-more-complex join algorithms without having the rest of the RDBMS. At some point we must build end-to-end systems to evaluate matching algorithms, to integrate R&D efforts, to educate our students in EM, and to make practical impacts.

In this aspect, EM can take inspiration from RDBMSs and Big Data systems. Pioneering systems such as System R, Ingres, and Hadoop have drastically helped push these fields forward, by helping to evaluate research ideas, providing an architectural blueprint for the entire community to focus on, facilitating more advanced systems, and making widespread real-world impacts.

The question then is what kinds of EM systems we should build, and how? In this paper we begin by showing that current EM systems suffer from four limitations that prevent them from being used extensively in practice.

First, when performing EM users often must execute many steps. Current systems however do not cover the entire EM pipeline, providing support for only a few steps (e.g., blocking, matching), while ignoring less well-known yet equally critical steps (e.g., debugging, sampling, cleaning).

Second, EM steps often must exploit many techniques, e.g., learning, mining, visualization, outlier detection, information extraction (IE), crowdsourcing, etc. Current EM systems (most of which are stand-alone monoliths that are not designed from scratch to "play well" with other systems) do not provide enough support for these techniques.

Third, users often have to write code to "patch" the system (e.g., to implement a lacking functionality or combine system components), ideally using a scripting environment, to enable rapid prototyping and iteration. Most current EM systems however do not provide such facilities.

Finally, in many EM scenarios users often do not know how to proceed end to end. Suppose a user wants to perform EM with at least 95% precision and 80% recall. Should he or she start out using a learning or rule-based EM approach? If learning-based, then which technique to select among the many existing ones? How to debug? What to do if after many tries the user still cannot reach 80% recall? Current EM systems provide no answers to such questions.

**The Magellan Solution:** To address these limitations, we describe Magellan, a new kind of EM systems currently being developed at UW-Madison, in collaboration with several industrial partners. Magellan (named after Ferdinand Magellan, the first end-to-end explorer of the globe) is novel in several important aspects.

First, Magellan focuses on helping power users (those who know how to code) execute a set of *EM scenarios* (e.g., using supervised learning to match two tables with a target accuracy). For each EM scenario, Magellan provides a comprehensive *how-to guide* that tells users what to do, step by step, end to end.

Second, Magellan identifies "pain points" in each guide, i.e., steps that require a lot of user effort, then provides tools to address those pain points. As we will see, these tools cover


DOI: https://doi.org/10.14778/2994509.2994535.

the entire EM pipeline (e.g., debugging, sampling), not just the blocking and matching steps.

Third, the tools are being built within the Python data science ecosystem, allowing users to easily exploit a wide range of techniques in learning, visualization, cleaning, etc. (as captured in numerous Python packages in this ecosystem, such as pandas, scikit-learn, matplotlib, pytorch, pyspark, etc.).

Finally, an added benefit of integration with the Python ecosystem is that Magellan is situated in a powerful scripting environment that users can use to prototype code to "patch" the system.

As described, Magellan assumes that the EM process cannot be automated. Instead it must involve the human user. So Magellan provides a detailed how-to guide that spells out where the human user must be involved and how, and where a tool can be used to reduce the user effort. Thus, Magellan is an example of *"human-in-the-loop" data management systems*, which have received significant recent attention [19].

**Challenges:** Realizing the above novelties raises major challenges. First, it turns out that developing effective how-to guides, even for very simple EM scenarios such as applying supervised learning to match, is already quite difficult, as we will show in Section 3.3.

Second, developing tools to support these guides is equally difficult. In particular, current EM work may have dismissed many steps in the EM pipeline as engineering. But here we show that many such steps (e.g., sampling, labeling, debugging, etc.) do raise difficult research challenges.

Finally, while most current EM systems are stand-alone monoliths, Magellan is designed to be placed within an "ecosystem" and is expected to "play well" with others (e.g., other Python packages). We say that Magellan is an "open-world system", because it relies on many other systems in the ecosystem in order to provide the fullest amount of support to the user doing EM. It turns out that building open-world systems raises non-trivial challenges, such as designing the right data structures and managing metadata.

**Current Status:** In the past three years we have started to address the above challenges. Specifically, we have open sourced Magellan [3]. As far as we can tell, Magellan is the most comprehensive open-source EM system today, in terms of the number of features it supports.

Magellan has been successfully used in five domain science projects at UW-Madison (in economics, biomedicine, environmental science [32, 33, 37, 9]), and at several companies (e.g., Johnson Control, Marshfield Clinic, Recruit Holdings [1], WalmartLabs). For example, at WalmartLabs it improved the recall of a deployed EM solution by 34%, while reducing precision slightly by 0.65%. It has also been used by 400+ students to match real-world data in five data science classes at UW-Madison (e.g., [2]).

Applying Magellan to the above real-world applications raised many research challenges. Examples include helping users finalize their matching definition [32, 19], debugging blocking [34], debugging rule-based EM [36], human-in-the-loop EM [19], applying deep learning to match textual data [35], hands-off string matching, data cleaning, and more. We have started to address some of these research challenges [34, 35, 36, 19], describe case studies [32], and summarize the lessons learned [19, 32]. Magellan and the data generated in this project have also been used by other research groups (e.g., [21, 25]).

**Table A**

| | Name | City | State |
|---|---|---|---|
| $a_1$ | Dave Smith | Madison | WI |
| $a_2$ | Joe Wilson | San Jose | CA |
| $a_3$ | Dan Smith | Middleton | WI |

**Table B**

| | Name | City | State |
|---|---|---|---|
| $b_1$ | David D. Smith | Madison | WI |
| $b_2$ | Daniel W. Smith | Middleton | WI |

| Matches |
|---|
| $(a_1, b_1)$ |
| $(a_3, b_2)$ |

**Figure 1: An example of matching two tables.**

**CloudMatcher and BigGorilla:** In terms of broader impacts, Magellan is an on-premise EM solution for power users. In a related project, we have been developing CloudMatcher, a hands-off cloud/crowd EM service for lay users (to be deployed soon at *cloudmatcher.io*) [27, 17, 26]. Our Magellan work has significantly influenced the development of CloudMatcher, by suggesting desired functionalities and pointing out possible limitations of such EM services [27].

The ideas underlying Magellan can potentially be applied to other types of data integration problems (e.g., schema matching, information extraction, data cleaning, etc.). We have started to flesh out a similar system-building agenda for data integration [20, 18]. We have also been partnering with Recruit Institute of Technology to encourage a community around BigGorilla, a repository of data preparation and integration tools [41]. The goal of BigGorilla is to foster an ecosystem of such tools, as a part of the Python data science ecosystem, for research, education, and practical purposes.

The rest of this paper motivates Magellan then discusses the solution architecture, empirical evaluation, lessons learned, and ongoing research directions. This paper is a condensed version of [29]. More details can be found in that paper and in [30, 31], and on the Magellan project's homepage [3].

## 2. LIMITATIONS OF CURRENT ENTITY MATCHING SYSTEMS

Entity matching (EM) has received much attention [13, 22]. A common EM scenario finds all tuple pairs that match, i.e., refer to the same real-world entity, between two tables $A$ and $B$ (see Figure 1). Other EM scenarios include matching tuples within a single table, matching into a knowledge base, matching XML data, etc. [13].

Most EM works have developed matching algorithms that exploit rules, learning, clustering, crowdsourcing, among others [13, 22]. The focus is on improving the matching accuracy and reducing costs (e.g., run time). Trying to match all pairs in $A \times B$ often takes very long. So users often employ heuristics to remove obviously non-matched pairs (e.g., products with different colors), in a step called *blocking*, before matching the remaining pairs. Several works have studied this step, focusing on scaling it up to large amounts of data (see Section 5).

In contrast to the extensive effort on matching algorithms, there has been relatively little work on building EM systems. As of 2016 we counted 18 major non-commercial systems (e.g., D-Dupe, DuDe, Febrl, Dedoop, Nadeef [13]), and 15 major commercial ones (e.g., Tamr, Data Ladder, IBM InfoSphere, IBM Midas [28, 38]). Our examination of these systems (see [30]) reveals the following four major problems:

**1. Systems Do Not Cover the Entire EM Pipeline:** When performing EM users often must execute many steps, e.g., blocking, matching, exploration, cleaning, extraction (IE), debugging, sampling, labeling, etc. Current systems provide support for only a few steps in this pipeline, while

ignoring less well-known yet equally critical steps.

For example, all 33 systems that we have examined provide support for blocking and matching. Twenty systems provide limited support for data exploration and cleaning. There is no meaningful support for any other steps (e.g., debugging, sampling, etc.). Even for blocking the systems merely provide a set of blockers that users can call; there is no support for selecting and debugging blockers, and for combining multiple blockers.

**2. Difficult to Exploit a Wide Range of Techniques:** Practical EM often requires a wide range of techniques, e.g., learning, mining, visualization, data cleaning, IE, SQL querying, crowdsourcing, keyword search, etc. For example, to improve matching accuracy, a user may want to clean the values of attribute "Publisher" in a table, or extract brand names from "Product Title", or build a histogram for "Price". The user may also want to build a matcher that uses learning, crowdsourcing, or some statistical techniques.

Current EM systems do not provide enough support for these techniques, and there is no easy way to do so. Incorporating all such techniques into a single system is extremely difficult. But the alternate solution of just moving data among a current EM system and systems that do cleaning, IE, visualization, etc. is also difficult and time consuming. A fundamental reason is that most current EM systems are stand-alone monoliths that are not designed from the scratch to "play well" with other systems. For example, many current EM systems were written in C, C++, C#, and Java, using proprietary data structures. Since EM is often iterative, we need to repeatedly move data among these EM systems and cleaning/IE/etc systems. But this requires repeated reading/writing of data to disk followed by complicated data conversion.

**3. Difficult to Write Code to "Patch" the System:** In practice users often have to write code, either to implement a lacking functionality (e.g., to extract product weights, or to clean the dates), or to tie together system components. It is difficult to write such code correctly in "one shot". Thus ideally such coding should be done using an interactive scripting environment, to enable rapid prototyping and iteration. This code often needs access to the rest of the system, so ideally the system should be in such an environment too. Unfortunately only 5 out of 33 systems provide such settings (using Python and R).

**4. Little Guidance for Users on How to Match:** In our experience this is by far the most serious problem with current EM systems. In many EM scenarios users simply do not know what to do: how to start, what to do next? Interestingly, even the simple task of taking a sample and labeling it (to train a learning-based matcher) can be quite complicated in practice, as we show in Section 3.3. Thus, it is not enough to just build a system consisting of a set of tools. It is also critical to provide step-by-step guidance to users on how to use the tools to handle a particular EM scenario and what to do when no tool is available. No EM system that we have examined provides such guidance.

# 3. THE MAGELLAN SOLUTION

We now describe Magellan and discuss how it addresses the above limitations. Figure 2 shows the Magellan architecture. The system targets a set of EM scenarios. For each EM scenario it provides a how-to guide. The guide proposes that the user solve the scenario in two stages: development and production.

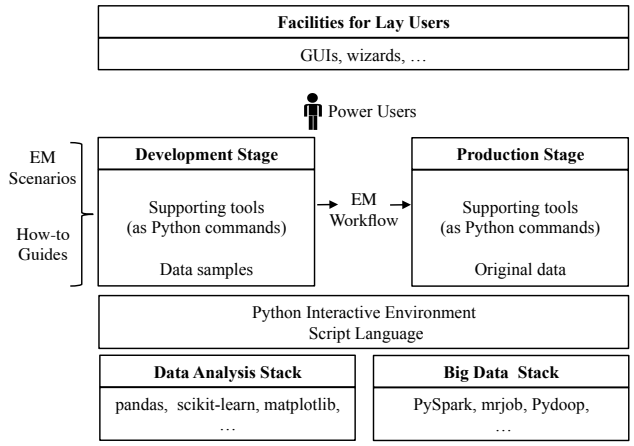



**Figure 2: The Magellan architecture.**

In the development stage, the user develops a good EM workflow (e.g., one with high matching accuracy). The guide tells the user what to do, step by step. For each step which is a "pain point", the user can use a set of supporting tools (each of which is a set of Python commands). This stage is typically done using data samples. In the production stage, the guide tells the user how to implement and execute the EM workflow on the entirety of data, again using a set of tools.

Both stages have access to the Python interactive scripting environment (e.g., Jupyter Notebook). Further, tools are built into the Python data science ecosystem. Thus, Magellan is an "open-world" system, as it often has to borrow functionalities (e.g., cleaning, extraction, visualization) from other Python packages in the ecosystem.

Finally, the current Magellan is geared toward power users (who can program). In the future facilities for lay users (e.g., GUIs, wizards) can be laid on top (see Figure 2), and lay user actions can be translated into sequences of commands in the underlying Magellan. In what follows we elaborate on the Magellan architecture.

## 3.1 EM Scenarios and Workflows

We classify EM scenarios along four dimensions: *(1) Problems:* Matching two tables; matching within a table; matching a table into a knowledge base; etc. *(2) Solutions:* Using learning; using learning and rules; performing data cleaning, blocking, then matching; performing IE, then cleaning, blocking, and matching; etc. *(3) Domains:* Matching two tables of biomedical data; matching e-commerce products given a large product taxonomy as background knowledge; etc. *(4) Performance:* Precision must be at least 92%, while maximizing recall as much as possible; both precision and recall must be at least 80%, and run time under four hours; etc.

An EM scenario can constrain multiple dimensions, e.g., matching two tables of e-commerce products using a rule-based approach with desired precision of at least 95%. Clearly there is a wide variety of EM scenarios. So we build Magellan to handle a few common scenarios, and then extend it to more scenarios over time. Specifically, for now we will consider the three scenarios that match two given relational tables $A$ and $B$ using (1) supervised learning, (2) rules, and

(3) learning plus rules, respectively. These scenarios are very common. In practice, users often try Scenario 1 or 2, and if neither works, then a combination of them (Scenario 3).

As discussed earlier, to handle an EM scenario, a user often has to execute many steps, such as cleaning, IE, blocking, matching, etc. The combination of these steps form an *EM workflow*. Figure 4 shows a sample workflow (which we explain in Section 3.3).

## 3.2 Development Stage vs. Production Stage

From our experience with real-world users doing EM, we propose that the how-to guide tell the user to solve the EM scenario in two stages: *development* and *production*. In the development stage the user finds a good EM workflow, typically using data samples. In the production stage the user applies the workflow to the entirety of data. Since this data is often large, a major concern here is to scale up the workflow. Other concerns include quality monitoring, logging, crash recovery, etc. The following example illustrates these two stages.

EXAMPLE 1. *Consider matching two tables $A$ and $B$ each having 1M tuples. Working with such large tables will be very time consuming in the development stage, especially given the iterative nature of this stage. Thus, in the development stage the user $U$ starts by sampling two smaller tables $A'$ and $B'$ from $A$ and $B$, respectively. Next, $U$ performs blocking on $A'$ and $B'$. The goal is to remove as many obviously non-matched tuple pairs as possible, while minimizing the number of matching pairs accidentally removed. $U$ may need to try various blocking strategies to come up with what he or she judges to be the best.*

*The blocking step can be viewed as removing tuple pairs from $A' \times B'$. Let $C$ be the set of remaining tuple pairs. Next, $U$ may take a sample $S$ from $C$, examine $S$, and manually write matching rules, e.g., "If titles match and the numbers of pages match then the two books match". $U$ may need to try out these rules on $S$ and adjust them as necessary. The goal is to develop matching rules that are as accurate as possible.*

*Once $U$ has been satisfied with the accuracy of the matching rules, the production stage begins. In this stage, $U$ executes the EM workflow that consists of the developed blocking strategy and matching rules on the original tables $A$ and $B$. To scale, $U$ may need to rewrite the code for blocking and matching to use Hadoop or Spark.* □

As described, these two stages are very different in nature: one goes for accuracy and the other goes for scaling (among others). Consequently, they will require very different sets of tools. We now discuss developing tools for these stages.

**Development Stage on a Data Analysis Stack:** We observe that what users try to do in the development stage is very similar in nature to data analysis tasks, which analyze data to discover insights. Indeed, creating EM rules can be viewed as analyzing (or mining) the data to discover accurate EM rules. Conversely, to create EM rules, users also often have to perform many data analysis tasks, e.g., cleaning, visualizing, finding outliers, IE, etc.

As a result, if we are to develop tools for the development stage in isolation, within a stand-alone monolithic system, as current work has done, we would need to somehow provide a powerful data analysis environment, in order for these tools to be effective. This is clearly very difficult to do.

So instead, we propose that tools for the development stage be developed on top of an open-source data analysis stack, so that they can take full advantage of all the data analysis tools already (or will be) available in that stack. In particular, two major data analysis stacks have recently been developed, based on R and Python. The Python stack for example includes the Python language, numpy and scipy packages for numerical/array computing, pandas for relational data management, scikit-learn for machine learning, among others. More tools are being added all the time. As of March 2018, there were 536 Python packages available in the popular Anaconda distribution. There is a vibrant community of contributors to continuously improve this stack.

For Magellan, since our initial target audience is the IT community, where we believe Python is more familiar, we have been developing tools for the development stage on the Python data analysis stack.

**Production Stage on a Big Data Stack:** In a similar vein, we propose that tools for the production stage, where scaling is a major focus, be developed on top of a Big Data stack. Magellan uses the Python Big Data stack, which consists of many software packages to run MapReduce (e.g., Pydoop, mrjob), Spark (e.g., PySpark), and parallel and distributed computing in general (e.g., pp, dispy).

In the rest of this paper we will focus on the development stage, leaving the production stage for subsequent papers.

## 3.3 How-to Guides and Tools

We now discuss developing how-to guides and tools to support these guides. First, we show that even for relatively simple EM scenarios (e.g., matching using supervised learning), a good guide can already be quite complex. Thus developing how-to guides is a major challenge, but such guides are critical in order to successfully guide the user through the EM process. Second, we show that each step of the guide, including those that prior work may have viewed as trivial (e.g., sampling, labeling), can raise many interesting research challenges.

Recall that Magellan currently targets three EM scenarios (Section 3.1). For space reasons, we will focus on the scenario of matching using supervised learning, and on developing a guide for the development stage of this scenario. Figure 3 shows the current version of this guide, listing only the top six steps. While each step may sound fairly informal (e.g., "create a set of features"), the full guide (available with Magellan's release) is far more complex and spells out in detail what to do (e.g., run a Magellan command to automatically create the features). We developed this guide based on observing how real-world users (e.g., at Walmart-Labs and Johnson Control) as well as students in several UW-Madison classes handled this scenario.

The guide states that to match two tables $A$ and $B$, the user should load the tables into Magellan (Step 1), do blocking (Step 2), label a sample of tuple pairs (Step 3), use the sample to iteratively find and debug a learning-based matcher (Steps 4-5), then return this matcher and its estimated matching accuracy (Step 6). We now briefly discuss these steps (see [30] for more details). For ease of exposition, we will assume that tables $A$ and $B$ share the same schema.

**Downsampling Tables:** We begin by loading the two tables $A$ and $B$ into memory. If these tables are large (e.g., each having 100K+ tuples), we should sample smaller tables $A'$ and $B'$ from $A$ and $B$ respectively, then do the develop-

1. Load tables A and B into Magellan. Downsample if necessary.

2. Perform blocking on the tables to obtain a set of candidate tuple pairs C.

3. Take a random sample S from C and label pairs in S as matched / non-matched.

4. Create a set of features then convert S into a set of feature vectors H. Split H into a development set I and an evaluation set J.

5. Repeat until out of debugging ideas or out of time:

   (a) Perform cross validation on I to select the best matcher. Let this matcher be X.

   (b) Debug X using I. This may change the matcher X, the data, labels, and the set of features, thus changing I and J.

6. Let Y be the best matcher obtained in Step 5. Train Y on I, then apply to J and report the matching accuracy on J.

**Figure 3: The top-level steps of the guide for the EM scenario of matching using supervised learning.**

ment stage with these smaller tables. Since this stage is iterative by nature, working with large tables can be very time consuming. Random sampling however does not work, because tables $A'$ and $B'$ may end up sharing very few matches. Thus we need a tool that samples more intelligently, to ensure a reasonable number of matches between $A'$ and $B'$. We have developed such a tool, which proved quite effective in our experiments (see [30]).

This tool however has a limitation: it may not get all important matching categories into $A'$ and $B'$. If so, the EM workflow created using $A'$ and $B'$ may not work well on the original tables $A$ and $B$. For example, consider matching companies. Tables $A$ and $B$ may contain two matching categories: (1) tuples with similar company names and addresses match because they refer to the same company, and (2) tuples with similar company names but different addresses may still match because they refer to different branches of the same company. Using the current tool, tables $A'$ and $B'$ may contain many tuple pairs of Case 1, but no or very few pairs of Case 2.

To address this problem, we are working on a better "downsampler". Our idea is to use clustering to create groups of matching tuples, then analyze these groups to infer matching categories, then sample from the categories. Major challenges here include how to effectively cluster tuples from the large tables $A$ and $B$, and how to define and infer matching categories accurately.

**Blocking to Create Candidate Tuple Pairs:** Next, we apply blocking to the tables $A'$ and $B'$ to generate a set $C$ of tuple pairs $(a \in A', b \in B')$. Many blocking solutions have been developed [13]. In practice, however, users often have three questions which current work has not addressed: (1) how to select the best blocker, (2) how to debug a given blocker, and (3) how to know when to stop?

*Selecting the Best Blocker:* There is no satisfactory solution yet to this problem. For now, based on our experience, we recommend that the user try successively more complex blockers. Specifically, the user can try overlap blocking first (e.g., "matching tuples must share at least $k$ tokens in an attribute $x$"), then attribute equivalence blocking (AE) (e.g., "matching tuples must share the same value for an attribute $y$"). These blockers are very fast, and can significantly cut down on the number of candidate tuple pairs. Next, the user can try other well-known blocking methods (e.g., sorted neighborhood, hash) if appropriate. Finally, the user can try rule-based blocking. This means the user can use multiple blockers and combine them in a flexible fashion (e.g., applying AE to the output of overlap blocking).

*Debugging Blockers:* Given a blocker $L$, how do we know if it does not remove too many matches? We have developed a debugger to answer this question [34]. Suppose applying $L$ to $A'$ and $B'$ produces a set $C$ of tuple pairs $(a \in A', b \in B')$. Then $D = A' \times B' \setminus C$ is the set of all tuple pairs removed by $L$. The debugger examines $D$ to return a list of $k$ tuple pairs in $D$ that are most likely to match. If the user $U$ finds many matches in the list, then that means blocker $L$ has removed too many matches. $U$ would need to modify $L$ to be less "aggressive", then apply the debugger again. Eventually if $U$ finds no or very few matches in the list, $U$ can assume that $L$ has removed no or very few matches, and thus is good enough.

*Knowing When to Stop Modifying the Blockers:* How do we know when to stop tuning a blocker $L$? Suppose applying $L$ to $A'$ and $B'$ produces the set of tuple pairs $block(L, A', B')$. The conventional wisdom is to stop when $block(L, A', B')$ fits into memory or is already small enough so that the matching step can process it efficiently.

In practice, however, this often does not work. For example, since we work with $A'$ and $B'$, *samples* from the original tables, monitoring $|block(L, A', B')|$ does not make sense. Instead, we want to monitor $|block(L, A, B)|$. But applying $L$ to the large tables $A$ and $B$ can be very time consuming, making the iterative process of tuning $L$ impractical.

As a result, users often want blockers that have (1) high pruning power, i.e., maximizing $1 - |block(L, A', B')|/|A' \times B'|$, and (2) high recall, i.e., maximizing the ratio of the number of matches in $block(L, A', B')$ divided by the number of matches in $A' \times B'$. Users can measure the pruning power, but so far they have had no way to estimate recall. This is where our debugger comes in. In our experiments (see Section 4) users reported they had used our debugger to find matches that the blocker $L$ had removed, and when they found no or only a few matches, they concluded that $L$ had achieved high recall and stopped tuning the blocker.

**Sampling and Labeling Tuple Pairs:** Let $L$ be the blocker we have created. Suppose applying $L$ to tables $A'$ and $B'$ produces a set of tuple pairs $C$. In the next step, user $U$ should take a sample $S$ from $C$, then label the pairs in $S$ as matched / no-matched, to be used later for training matchers, among others.

At a first glance, this step seems simple: why not just take a random sample and label it? Unfortunately in practice this is far more complicated. For example, suppose $C$ contains relatively few matches (either because there are few matches between $A'$ and $B'$, or because blocking was too liberal, resulting in a large $C$). Then a random sample $S$ from $C$ may contain no or few matches. But the user $U$ often does not recognize this until $U$ has labeled most of the pairs in $S$. This is a waste of $U$'s time and can be quite serious in cases where labeling is time consuming or requires expensive domain experts (e.g., labeling drug pairs when we worked with Marshfield Clinic). We have developed a solution to address this problem, building on the work in [26] (see [30]).

**Selecting a Matcher:** Once user $U$ has labeled a sample $S$, $U$ uses $S$ to select a good initial learning-based matcher. Our guide provides a tool to address this problem. The tool

first automatically generates a set of features, uses them to convert each pair in $S$ into a feature vector, then performs cross validation over a subset of the feature vectors to select the matcher with the highest estimated accuracy from among those supplied by Magellan.

**Debugging a Matcher:** Let the selected matcher be $X$. Next, user $U$ debugs $X$ to improve its accuracy. Such debugging is critical in practice, yet has received little attention. Our guide suggests that user $U$ debug in three steps: (1) identify and understand the matching mistakes made by $X$, (2) categorize these mistakes, and (3) take actions to fix common categories of mistakes.

*Identifying and Understanding Matching Mistakes:* Given a labeled set $I$ for debugging purpose, $U$ should split $I$ into two sets $P$ and $Q$, train $X$ on $P$ then apply it to $Q$ to identify the matching mistakes made by $X$ in $Q$ (this process can be repeated many times, using different $P$ and $Q$). These are *false positives* (non-matching pairs predicted matching) and *false negatives* (matching pairs predicted not). Addressing them helps improve precision and recall, respectively.

Next $U$ should try to understand why $X$ makes each mistake, using a match debugger where available. There are four major categories of mistakes. (1) The data can be dirty, e.g., the price value is incorrect. (2) The label can be wrong, e.g., a pair should have been labeled "not matched". (3) The feature set is problematic. A feature is misleading, or a new feature is desired, e.g., we need a new feature that extracts and compares the publishers. (4) The learning algorithm employed by $X$ is problematic, e.g., a parameter such as "maximal depth to be searched" is set to be too small. Currently Magellan has debuggers for a set of learning-based matchers, e.g., decision tree, random forest. We are working on improving these debuggers and developing debuggers for more learning algorithms.

*Categorizing Matching Mistakes:* After $U$ has examined all or a large number of matching mistakes, he or she can categorize them, based on problems with data, label, feature, and the learning algorithm. Examining all or most mistakes is very time consuming. Thus a consistent feedback we have received from real-world users is that they would love a tool that can automatically examine and give a preliminary categorization of the types of the matching mistakes. As far as we can tell, no such tool exists today.

*Handling Common Categories of Mistakes:* Next $U$ should try to fix common categories of mistakes by modifying the data, labels, set of features, and the learning algorithm. This part often involves data cleaning and extraction (IE), e.g., normalizing all values of attribute "affiliation", or extracting publishers from attribute "desc" then creating a new feature comparing the publishers.

This part is often also very time consuming. Real-world users have consistently indicated needing support in at least two areas. First, they want to know exactly what kinds of data cleaning and IE operations they need to do to fix the mistakes. Naturally they want to do as minimally as possible. Second, re-executing the entire EM process after each tiny change to see if it "fixes" the mistakes is very time consuming. Hence, users want an "what-if" tool that can quickly show the effect of a hypothetical change.

**The Resulting EM Workflow:** After executing the above steps, user $U$ has in effect created an EM workflow,

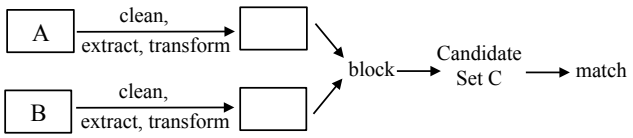



**Figure 4: The EM workflow for the learning-based matching scenario.**

as shown in Figure 4. Since this workflow will be used in the production stage, it takes as input the two original tables $A$ and $B$. Next, it performs a set of data cleaning, IE, and transformation operations on these tables. These operations are derived from the debugging step discussed in Section 3.3. Next, the workflow applies the blockers created in Section 3.3 to obtain a set of candidate tuple pairs $C$. Finally, the workflow applies the learning-based matcher created in Section 3.3 to the pairs in $C$.

Note that the steps of sampling and labeling a sample $S$ do not appear in this workflow, because we need them only in the development stage, in order to create, debug, and train matchers. Once we have found a good learning-based matcher (and have trained it using $S$), we do not have to execute those steps again in the production stage.

# 4. EMPIRICAL EVALUATION

As of March 2018, Magellan [3] consists of 6 Python packages, 37K lines of code, and 231 commands. It has been developed over 3 years by 13 developers. We now describe using Magellan in data science classes, with domain scientists at UW-Madison, and at companies.

## 4.1 Using Magellan in Data Science Classes

So far 400+ students (including 90+ undergraduates) have used Magellan in 5 data science classes at UW-Madison. These students can be considered the equivalents of power users at organizations. They know Python but are not experts in EM. We asked them to form team of 2-3 students, then asked each team to find two data-rich Web sites, extract and convert data from them into two relational tables, then apply Magellan to match tuples across the tables [16]. We typically asked each team to do the EM scenario of supervised learning followed by rules, and aim for precision of at least 90% with recall as high as possible (a very common scenario in practice).

We now describe in more details our experience with a Fall 2015 class, which consisted of 44 students divided into 24 teams (see [30] for details). These teams extracted tables in 12 domains (e.g., Vehicles, Movies, Restaurants, Music, etc.). The tables have 7,313 tuples on average, with 5-17 attributes. On these tables, the best learning-based matcher (after cross validation) achieved accuracy P = 56-100%, R = 37.5-100%, $F_1$ = 56-99.5%, suggesting that many of these tables are not easy to match. Using Magellan, however, the teams were able to significantly improve these accuracies, achieving P = 91.3-100%, R = 64.7-100%, $F_1$ = 78.6-100%. All 24 teams achieved precision exceeding 90%, and 20 teams also achieved recall exceeding 90%. (Four teams had recall below 90% because their data were quite dirty, with many missing values.) All teams reported being able to follow the how-to guide. Together with qualitative feedback from the teams, this suggests that users can follow Magellan's how-to guide to achieve high matching accuracy on diverse data sets.

All teams used 1-5 blockers (e.g., attribute equivalence, overlap, rule-based), for an average of 3. On average 3 different types of blockers were used per team. This suggests that it is relatively easy to create a blocking pipeline with diverse blocker types. All teams debugged their blockers, in 1-10 iterations, for an average of 5. 18 out of 24 teams used our debugger [34], and reported that it helped in four ways: cleaning data, finding the correct blocker types and attributes, tuning blocker parameters, and knowing when to stop (see [30]). Teams reported spending 4-32 hours on blocking (including reading documentations). Overall, 21 out of 24 teams were able to prune away more than 95% of $|A \times B|$, with an average reduction of 97.3%, suggesting that they were able to construct blocking with high pruning rate.

Recall from Section 3.3 that after cross validation to select the best learning-based matcher $X$, user $U$ iteratively debugged $X$ to improve its accuracy. Teams performed 1-5 debugging iterations, for an average of 3. They added and deleted features, cleaned data based on the debugging result, and tuned the parameters of the learning algorithm. These actions helped improve accuracies from 56-100% to 73.3-100% precision, and 37.5-100% to 61-100% recall. Adding rules further improves accuracy: precision from 73.3-100% to 91.3-100% and recall from 61-100% to 64.7-100%.

## 4.2 Domain Sciences and Companies

So far Magellan has been applied to five projects in three domain sciences at UW-Madison. First, a team of applied economists used Magellan to match two tables of 1,832 and 1,916 grant descriptions, respectively [32]. Magellan achieved significantly better accuracy, improving recall by 23% while achieving comparable precision, compared to a rule-based EM solution deployed at [32]. The same team also used Magellan to match two tables of 1,851 and 13.5M organization descriptions, respectively.

A team in biomedicine used Magellan to match two tables of 453K and 451K of drug descriptions, achieving 99.1% precision and 95.2% recall [33, 37]. Another team in biomedicine used Magellan to match attribute names within a community data repository [9]. Finally, a team in environmental sciences also used Magellan to match attribute names within a community data repository. These last two examples show how Magellan can also be used to match schema elements, not just data instances.

Magellan has also been used for EM at several companies, including WalmartLabs, Johnson Control, Marshfield Clinic, and Recruit Holdings. At WalmartLabs, Magellan was able to help improve the recall of a deployed EM solution by 34% while reducing precision slightly by 0.65%. Johnson Control has used Magellan to match addresses (between tables of size 90K vs. 231K) and vendors (within a single table of size 50K). Marshfield Clinic was involved in the drug matching project described earlier [33, 37]. Recruit Holdings used Magellan to match stores, companies, and properties (e.g., de-duplicating 10K store names with 98.9% accuracy) [1].

## 4.3 Discussion

Our experience with Magellan suggest that users can successfully follow the how-to guide to achieve high EM accuracy on diverse data sets. In fact, we consider the how-to guide to be the single most important component of the system. Without it, users are lost: they do not even know where to start, when to use what tools, and how.

Our experience further suggests that the various tools developed for Magellan (e.g., debuggers) can be highly effective in helping the users. It also clearly shows that practical EM requires a wide range of capabilities, e.g., cleaning, extraction, visualization, underscoring the importance of placing Magellan in an ecosystem that provides such capabilities. (In fact, Magellan currently uses 11 packages in the Python ecosystem to provide such capabilities.)

At the same time, our experience also raises many interesting challenges. First, it turns out that at the start of an EM project, users often do not know what it means to match, i.e., there are often many alternative match definitions, and users often are not even aware of these, let alone selecting the right one [19, 32]. This can significantly complicate the EM process. Thus, it is highly desirable to have a step in the how-to guide (together with some tools) to help users explore and finalize the match definition. Second, some users want to play around with multiple match definitions, just to see how sensitive to these definitions the inferences based on the matches are. Third, a portion of the data may turn out to be so dirty for EM that it should be removed before continuing with the EM process, but how can we detect such portions? Fourth, an EM team is often geographically distributed. How can they use Magellan in such settings. Finally, Magellan is an "open-world system", in that it relies on many other packages in the Python ecosystem in order to provide the fullest amount of support to the user doing EM. It turns out that building open-world systems raises non-trivial challenges, such as designing the right data structures and managing metadata [30]. There are many other challenges (e.g., how to debug and serve learning models, how to visualize the matches, etc.). In recent papers we have tried to summarize some of these case studies, lessons learned, and challenges [32, 19, 29, 27, 35]. We have also started to address some of these challenges [34, 35, 36, 19]. But much more remains to be done.

# 5. RELATED WORK

Numerous EM algorithms have been proposed [13, 22]. But far fewer EM systems have been developed. We discussed these systems in Section 2 (see also [13]). For matching using supervised learning (Section 3.3), some of these systems provide only a set of matchers. None provides support for sampling, labeling, selecting and debugging blockers and matchers, as Magellan does.

Some recent works have discussed desirable properties for EM systems, e.g., being extensible and easy-to-deploy [15], being flexible and open source [12], and the ability to construct complex EM workflow consisting of distinct phases, each requiring a specific technique depending on the given application and data requirements [23]. The IBM Midas project has also proposed a language for helping users tackle the different stages of the EM pipeline [28, 38]. These works however do not discuss covering the entire EM pipeline, how-to guides, building on top of data analysis and Big Data stacks, and open-world systems, as we do in this paper.

Several works have addressed scaling up blocking, learning blockers, and using crowdsourcing for blocking (see [14] for a survey). As far as we know, there has been no work on debugging blocking, as we do in Magellan (see [34]).

On sampling and labeling, several works have studied active sampling [39, 6, 8]. These methods however are not directly applicable in our context, where we need a representative sample in order to estimate the matching accuracy

(see Step 6 in Figure 3). For this purpose our work is closest to [26], which uses crowdsourcing to sample and label.

Debugging learning models has received relatively little attention, even though it is critical in practice, as this paper has demonstrated. Prior works help users build, inspect and visualize specific ML models (e.g., decision trees [5], Naive Bayes [7], SVM [11], ensemble model [40]). But they do not allow users to examine errors and inspect raw data. In this aspect, the work closest to ours is [4], which addresses iterative building and debugging of supervised learning models. The system proposed in [4] can potentially be implemented as a Magellan's tool for debugging learning-based matchers.

Finally, the notion of "open world" has been discussed in [24], but in the context of crowd workers' manipulating data inside an RDBMS. Here we discuss a related but different notion of open-world systems that often interact with and manipulate each other's data. In this vein, the work [10] is related in that it discusses the API design of the scikit-learn package and its design choices to seamlessly tie in with other packages in Python.

# 6. CONCLUSIONS & ONGOING WORK

We have argued that significantly more attention should be paid to building EM systems. We described Magellan, a new kind of EM systems, which is novel in several important aspects: how-to guides, tools to support the entire EM pipeline, tight integration with the PyData ecosystem, open world vs. closed world systems, and easy access to an interactive script environment.

We are conducting more real-world evaluation of Magellan, further examining the research challenges raised in this paper, and extending Magellan with more capabilities (e.g., crowdsourcing). Building on Magellan, we have also been working on two other projects. CloudMatcher is a cloud/crowd EM service for lay users [27, 17, 26]. The how-to guide of Magellan helps us determine which capabilities to add to CloudMatcher, to make it useful in performing EM end to end [27]. BigGorilla is a joint effort led by UW-Madison and Recruit Institute of Technology to encourage a community around an open-source ecosystem of data preparation and integration tools [41]. Currently, BigGorilla curates tools for schema matching, information extraction, and entity matching (including Magellan), among others.

**Acknowledgment:** We thank the SIGMOD Record's associate editors for shepherding this paper. This work is generously supported by WalmartLabs, Google, Johnson Control, American Family Insurance, UW-Madison UW2020 grant, NIH BD2K grant U54 AI117924, and NSF Medium grant IIS-1564282.

# 7. REFERENCES

[1] BigGorilla: An Open-source Data Integration and Data Preparation Ecosystem: https://recruit-holdings.com/news_data/release/2017/0630_7890.html.

[2] CS 838: Data Science: Principles, Algorithms, and Applications https://sites.google.com/site/anhaidgroup/courses/cs-838-spring-2017/project-description/stage-3.

[3] Magellan home page https://sites.google.com/site/anhaidgroup/projects/magellan.

[4] S. Amershi et al. Modeltracker: Redesigning performance analysis tools for machine learning. CHI, 2015.

[5] M. Ankerst et al. Visual classification: An interactive approach to decision tree construction. KDD, 1999.

[6] A. Arasu, M. Götz, and R. Kaushik. On active learning of record matching packages. SIGMOD, 2010.

[7] B. Becker, R. Kohavi, and D. Sommerfield. Visualizing the simple Bayesian classifier. In *Information Visualization in Data Mining and Knowledge Discovery*, 2002.

[8] K. Bellare, S. Iyengar, A. G. Parameswaran, and V. Rastogi. Active sampling for entity matching. KDD, 2012.

[9] M. Bernstein et al. MetaSRA: normalized human sample-specific metadata for the sequence read archive. *Bioinformatics*, 33(18):2914–2923, 2017.

[10] L. Buitinck et al. API design for machine learning software: experiences from the scikit-learn project. *arXiv preprint arXiv:1309.0238*, 2013.

[11] D. Caragea, D. Cook, and V. Honavar. Gaining insights into support vector machine pattern classifiers using projection-based tour methods. KDD, 2001.

[12] P. Christen. Febrl: A freely available record linkage system with a graphical user interface. HDKM, 2008.

[13] P. Christen. *Data Matching*. Springer, 2012.

[14] P. Christen. A survey of indexing techniques for scalable record linkage and deduplication. *TKDE*, 24(9):1537–1555, 2012.

[15] M. Dallachiesa et al. Nadeef: A commodity data cleaning system. SIGMOD, 2013.

[16] S. Das et al. The Magellan data repository. https://sites.google.com/site/anhaidgroup/projects/data.

[17] S. Das et al. Falcon: Scaling up hands-off crowdsourced entity matching to build cloud services. In *SIGMOD*, 2017.

[18] A. Doan. What is our agenda for data science? In *CIDR*, 2017.

[19] A. Doan et al. Human-in-the-loop challenges for entity matching: A midterm report. In *HILDA*, 2017.

[20] A. Doan et al. Toward a system building agenda for data integration and cleaning. In *IEEE Data Engineering Bulletin, Special Issue on Data Integration (to appear)*, 2018.

[21] M. Ebraheem et al. DeepER–deep entity resolution. *arXiv preprint arXiv:1710.00597*, 2017.

[22] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios. Duplicate record detection: A survey. *IEEE TKDE*, 19(1):1–16, 2007.

[23] M. Fortini et al. Towards an open source toolkit for building record linkage workflows. In *In SIGMOD Workshop on Information Quality in Information Systems*, 2006.

[24] M. J. Franklin et al. CrowdDB: answering queries with crowdsourcing. SIGMOD, 2011.

[25] C. Ge et al. Private exploration primitives for data cleaning. *arXiv preprint arXiv:1712.10266*, 2017.

[26] C. Gokhale et al. Corleone: Hands-off crowdsourcing for entity matching. SIGMOD, 2014.

[27] Y. Govind et al. Cloudmatcher: A cloud/crowd service for entity matching. In *BIGDAS*, 2017.

[28] M. A. Hernández et al. HIL: a high-level scripting language for entity integration. In *EDBT*, 2013.

[29] P. Konda et al. Magellan: Toward building entity matching management systems. *PVLDB*, 9(12):1197–1208, 2016.

[30] P. Konda et al. Magellan: Toward building entity matching management systems. 2016. Technical Report, http://www.cs.wisc.edu/~anhai/papers/magellan-tr.pdf.

[31] P. Konda et al. Magellan: Toward building entity matching management systems over data science stacks. *PVLDB*, 9(13):1581–1584, 2016.

[32] P. Konda et al. Performing entity matching end to end: A case study. 2016. Technical Report, http://www.cs.wisc.edu/~anhai/papers/umetrics-tr.pdf.

[33] E. LaRose et al. Entity matching using Magellan: Mapping drug reference tables. In *AIMA Joint Summit*, 2017.

[34] H. Li et al. Matchcatcher: A debugger for blocking in entity matching. In *EDBT*, 2018.

[35] S. Mudgal et al. Deep learning for entity matching: A design space exploration. In *SIGMOD*, 2018.

[36] F. Panahi et al. Towards interactive debugging of rule-based entity matching. In *EDBT*, 2017.

[37] P. Pessig. Entity matching using Magellan - Matching drug reference tables. In CPCP Retreat 2017. http://cpcp.wisc.edu/resources/cpcp-2017-retreat-entity-matching.

[38] K. Qian et al. Active learning for large-scale entity resolution. In *CIKM*, 2017.

[39] S. Sarawagi and A. Bhamidipaty. Interactive deduplication using active learning. KDD, 2002.

[40] J. Talbot et al. Ensemblematrix: Interactive visualization to support machine learning with multiple classifiers. CHI, 2009.

[41] W.-C. Tan et al. Big gorilla: an open-source ecosystem for data preparation and integration. In *IEEE Data Engineering Bulletin, Special Issue on Data Integration (to appear)*, 2018.

# Technical Perspective: Natural Language Explanations for Query Results

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

Motivated by conversational agents such as Siri, Cortana, the Google Assistant, and Alexa — there has been a surge of interest in spoken as well as textual natural language interfaces. To this point, such systems have relied on innovations in speech recognition (such as recurrent neural networks, LSTMs, and so on) and in specially encoding specific question-answering strategies via "skills." A "natural" question for the SIGMOD community is how to best connect natural language interfaces systems to DBMSs, ideally in a way that generalizes to any database schema or instance.

In fact, the problem of providing a natural language interface to a database system (i.e., mapping from a spoken or textual question to a structured query) dates back at least to the 1980s [4]. Such efforts had middling success due to issues of accuracy, so the problems were later revisited in the 2000's with an eye towards restricting the space of options in order to improve precision [6]. Nonetheless, such systems did not gain much traction, again due to the challenges of ensuring accuracy for a given database when the user might ask an ambiguous question.

Recent work by Li and Jagadish [5], called `NaLIR`, proposed an *interactive communicator* within the query system, which presents to the user a *query tree* explaining what the system was going to do — such that the user could correct any mistakes. This was helpful in improving reliability, but it required that the user understand tree structured representations of queries.

In "Natural Language Explanations for Query Results," Deutch and his co-authors suggest that a more effective means of helping the user understand and correct results might be through *provenance* information — i.e., giving an explanation for each answer of how and why it exists. Their approach adapts the `NaLIR` system and nicely leverages the recent body of work on provenance semirings [3, 2, 1]. The provenance semiring model has an important property that equivalent query plans (as produced by a query optimizer) will have equivalent provenance expressions.

The innovations in this paper are in three areas. First, the authors use *the structure of the natural language query itself* (and the mappings to structured queries, and then later, from queries to provenance) to present the provenance in a form that matches the natural language query — and thus the user's expectations. Second, they reduce the size (and repetition) of the provenance via factoring. Finally, they incorporate aggregate results (e.g., counts) in place of certain details.

The paper does a great job of clearly identifying and articulating what makes the provenance problem different for natural language query systems, and presenting elegant technical solutions to these new challenges.

## 1. REFERENCES

[1] Yael Amsterdamer, Daniel Deutch, and Val Tannen. Provenance for aggregate queries. In *PODS*, pages 153–164, 2011.

[2] Todd J. Green, Grigoris Karvounarakis, Zachary G. Ives, and Val Tannen. Update exchange with mappings and provenance. In *VLDB*, 2007. Amended version available as Univ. of Pennsylvania report MS-CIS-07-26.

[3] Todd J. Green, Grigoris Karvounarakis, and Val Tannen. Provenance semirings. In *PODS*, 2007.

[4] S. Jerrold Kaplan. Designing a portable natural language database query system. *ACM Trans. Database Syst.*, 9:1–19, March 1984. Available from http://doi.acm.org/10.1145/348.318584.

[5] Fei Li and HV Jagadish. Constructing an interactive natural language interface for relational databases. *Proceedings of the VLDB Endowment*, 8(1):73–84, 2014.

[6] Ana-Maria Popescu, Oren Etzioni, and Henry Kautz. Towards a theory of natural language interfaces to databases. In *IUI*, pages 327–327, 2003.

# Natural Language Explanations for Query Results

Daniel Deutch
Tel Aviv University
danielde@post.tau.ac.il

Nave Frost
Tel Aviv University
navefrost@mail.tau.ac.il

Amir Gilad
Tel Aviv University
amirgilad@mail.tau.ac.il

## ABSTRACT

Multiple lines of research have developed Natural Language (NL) interfaces for formulating database queries. We build upon this work, but focus on presenting a highly detailed form of the *answers* in NL. The answers that we present are importantly based on the *provenance* of tuples in the query result, detailing not only the results but also their *explanations*. We develop a novel method for transforming provenance information to NL, by leveraging the original NL query structure. Furthermore, since provenance information is typically large and complex, we present two solutions for its effective presentation as NL text: one that is based on provenance factorization, with novel desiderata relevant to the NL case, and one that is based on summarization.

## 1. INTRODUCTION

Developing Natural Language (NL) interfaces to database systems has been the focus of multiple lines of research (see *e.g.* [17, 2, 21]). In this work we complement these efforts by providing *NL explanations to query answers*. The explanations that we provide elaborate upon answers with additional important information, and are helpful for understanding *why* does each answer qualify to the query criteria.

As an example, consider the Microsoft Academic Search database (`http://academic.research.microsoft.com`) and consider the NL query in Figure 1a. A state-of-the-art NL query engine, `NaLIR` [17], is able to transform this NL query into the SQL query also shown (as a Conjunctive Query, which is the fragment that we focus on in this paper) in Figure 1b. When evaluated using a standard database engine, the query returns the expected list of organizations. However, the answers (organizations) in the query result lack *justification*, which in this case would include the authors affiliated with each organization and details of the papers they have published (their titles, their publication venues and publication years). Such additional information, corresponding to the notion of *provenance* (*e.g.* [12, 14, 6]) can lead to a richer answer than simply providing the names of organizations: it allows users to also see relevant details of the qualifying organizations. Provenance information is also valuable for validation of answers: a user who sees an organization name as an answer is likely to have a harder time validating that this organization qualifies as an answer, than if she was presented with the full details of publications.

```
return the organization of authors who published papers
in database conferences after 2005
```

**(a)** NL Query

```
query(oname) :- org(oid, oname), conf(cid, cname),
pub(wid, cid, ptitle, pyear), author(aid, aname, oid),
domainConf(cid, did), domain(did, dname),
writes(aid, wid), dname = 'Databases', pyear > 2005
```

**(b)** CQ $Q$

**Figure 1:** NL Query and CQ $Q$

```
TAU is the organization of Tova M. who published
'OASSIS...' in SIGMOD in 2014
```

**Figure 2:** Answer For a Single Assignment

We propose a novel approach of presenting *provenance information for answers of NL queries, again as sentences in Natural Language*. Continuing our running example, Figure 2 shows one of the answers outputted by our system in response to the NL query in Figure 1a.

Our solution consists of the following key contributions.

*Provenance Tracking Based on the NL Query Structure.* A first key idea in our solution is to leverage the *NL query structure* in constructing NL provenance. In particular, we modify `NaLIR` so that we store exactly which parts of the NL query translate to which parts of the formal query. Then, we evaluate the formal query using a provenance-aware engine (we use `SelP` [7]), further modified so that it stores which parts of the query "contribute" to which parts of the provenance. By composing these two "mappings" (text-to-query-parts and query-parts-to-provenance) we infer which parts of the NL query text are related to which provenance parts. Finally, we use the latter information in an "inverse" manner, to translate the provenance to NL text.

*Factorization.* A second key idea is related to the provenance size. In typical scenarios, a single answer may have multiple explanations (multiple authors, papers, venues and years in our example). A naïve solution is to formulate and present a separate sentence corresponding to each explana-

 DOI: https://doi.org/10.14778/3055540.3055550

We are extremely grateful to Fei Li and H.V. Jagadish for generously sharing with us the source code of `NaLIR`, and providing invaluable support.

tion. The result will however be, in many cases, very long and repetitive. As observed already in previous work [4, 18], different assignments (explanations) may have significant parts in common, and this can be leveraged in a *factorization* that groups together multiple occurrences. In our example, we can *e.g.* factorize explanations based on author, paper name, conference name or year. Importantly, we impose a novel constraint on the factorizations that we look for (which we call *compatibility*), intuitively capturing that their structure is consistent with a partial order defined by the parse tree of the question. This constraint is needed so that we can translate the factorization back to an NL answer whose structure is similar to that of the question. Even with this constraint, there may still be exponentially many (in the size of the provenance expression) compatible factorizations, and we look for the factorization with minimal size out of the compatible ones; for comparison, previous work looks for the minimal factorization with no such "compatibility constraint". The corresponding decision problem remains coNP-hard (again in the provenance size), but we devise an effective and simple greedy solution. We further translate factorized representations to concise NL sentences, again leveraging the structure of the NL query.

*Summarization.* We propose *summarized* explanations by replacing details of different parts of the explanation by their synopsis, *e.g.* presenting only the number of papers published by each author, the number of authors, or the overall number of papers published by authors of each organization. Such summarizations incur by nature a loss of information but are typically much more concise and easier for users to follow. Here again, while provenance summarization has been studied before (*e.g.* [1, 18]), the desiderata of a summarization needed for NL sentence generation are different, rendering previous solutions inapplicable here. We observe a tight correspondence between factorization and summarization: every factorization gives rise to multiple possible summarizations, each obtained by counting the number of sub-explanations that are "factorized together". We provide a robust solution, allowing to compute NL summarizations of the provenance, of varying levels of granularity.

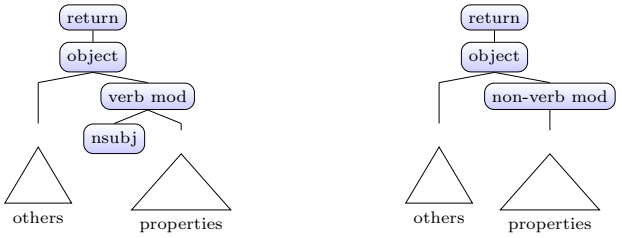
## 2. PRELIMINARIES

We provide here the necessary preliminaries on Natural Language Processing, conjunctive queries and provenance.

### 2.1 From NL to Formal Queries

We start by recalling some basic notions from NLP, as they pertain to the translation process of NL queries to a formal query language. A key notion that we will use is that of the *syntactic dependency tree* of NL queries. This is essentially a node-labeled tree where labels consist of two components, as follows: (1) Part of Speech ($POS$): the syntactic role of the word; (2) Relationship ($REL$): the grammatical relationship between the word and its parent in the dependency tree.

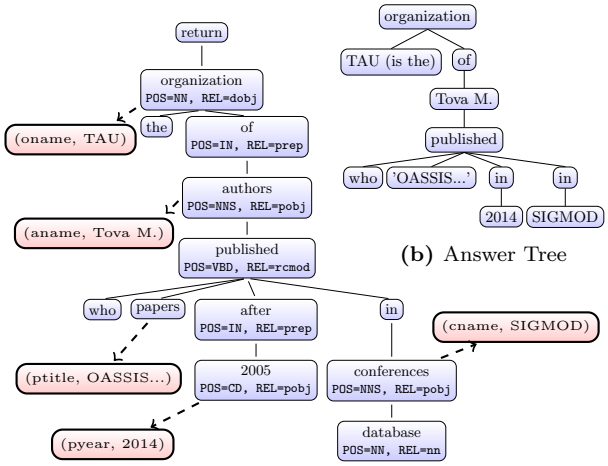
We focus on a sub-class of queries handled by `NaLIR`, namely that of Conjunctive Queries, possibly with comparison operators ($=, >, <$) (`NaLIR` further supports nested queries and aggregation). The corresponding NL queries in `NaLIR` follow one of the two (very general) abstract forms described in Figure 3: an object (noun) is sought for, that satisfies some properties, possibly described through a complex sub-




**(a)** Verb Mod. **(b)** Non-Verb Mod.

**Figure 3:** Abstract Dependency Trees

sentence rooted by a *modifier* (which may or may not be a verb, a distinction whose importance will be made clear later).




**(a)** Query Tree **(b)** Answer Tree

**Figure 4:** Question and Answer Trees

Example 2.1. *Reconsider the NL query in Figure 1a; its dependency tree is depicted in Figure 4a (ignore for now the arrows). The part-of-speech (POS) tag of each node reflects its syntactic role in the sentence – for instance, "organization" is a noun (denoted "NN"), and "published" is a verb in past tense (denoted "VBD"). The relation (REL) tag of each node reflects the semantic relation of its sub-tree with its parent. For instance, the REL of "of" is prep ("prepositional modifier") meaning that the sub-tree rooted at "of" describes a property of "organization" while forming a complex sub-sentence. The tree in Figure 4a matches the abstract tree in Figure 3b since "organization" is the object and "of" is a non-verb modifier (its POS tag is IN, meaning "preposition or subordinating conjunction") rooting a sub-sentence describing "organization".*

The dependency tree is transformed by `NaLIR`, based also on schema knowledge, to SQL. We focus in this work on NL queries that are compiled into Conjunctive Queries (CQs).

Example 2.2. *Reconsider our running example NL query in Figure 1a; a counterpart Conjunctive Query is shown in Figure 1b. Some words of the NL query have been mapped by* `NaLIR` *to variables in the query, e.g., the word "organization" corresponds to the head variable (oname). Additionally, some parts of the sentence have been complied*

```
(oname,TAU)·(aname,Tova M.)·(ptitle,OASSIS...)·
(cname,SIGMOD)·(pyear,14')+
(oname,TAU)·(aname,Tova M.)·(ptitle,Querying...)·
(cname,VLDB)·(pyear,06')+
(oname,TAU)·(aname,Tova M.)· (ptitle,Monitoring..)·
(cname,VLDB)·(pyear,07')+
(oname,TAU)·(aname,Slava N.)·(ptitle,OASSIS...)·
(cname, SIGMOD)·(pyear,14')+
(oname,TAU)·(aname,Tova M.)·(ptitle,A sample...)·
(cname,SIGMOD)·(pyear,14')+
(oname,UPENN)·(aname,Susan D.)·(ptitle,OASSIS...)·
(cname,SIGMOD)·(pyear,14')
```

**Figure 5:** Value-level Provenance

*to boolean conditions based on the MAS schema, e.g., the part "in database conferences" was translated to dname $='$ Databases$'$ in the CQ depicted in Figure 1b. Figure 4a shows the mapping of some of the nodes in the NL query dependency tree to variables of Q (ignore for now the values next to these variables).*

The translation performed by `NaLIR` from an NL query to a formal one can be captured by a *mapping* from (some) parts of the sentence to parts of the formal query. It can also be defined as a partial function from the nodes of the dependency tree to the variables of the query. We denote it by dependency-to-query-mapping.

## 2.2 Provenance

After compiling a formal query corresponding to the user's NL query, we evaluate it and keep track of *provenance*, to be used in explanations. To define provenance, we first exemplify the standard notion of *assignments* for CQs.

Assignments allow for defining the semantics of CQs: a tuple $t$ is said to appear in the query output if there exists an assignment $\alpha$ s.t. $t = \alpha(head(Q))$. They will also be useful in defining provenance below.

Example 2.3. *Consider again the query Q in Figure 1b and the database in Figure 6. The tuple (TAU) is an output of Q when assigning the highlighted tuples to the atoms of Q. As part of this assignment, the tuple (2, TAU) (the second tuple in the org table) and (4, Tova M., 2) (the second tuple of the author table) are assigned to the first and second atom of Q, respectively. In addition to this assignment, there are 4 more assignments that produce the tuple (TAU) and one assignment that produces the tuple (UPENN).*

We next leverage assignments in defining provenance, introducing a simple value-level model. The idea is that assignments capture the *reasons* for a tuple to appear in the query result, with each assignment serving as an *alternative* such reason (indeed, the existence of a single assignment yielding the tuple suffices, according to the semantics, for its inclusion in the query result). Within each assignment, we keep record of the value assigned to each variable, and note that the *conjunction* of these value assignments is required for the assignment to hold. Capturing alternatives through the symbolic " $+$ " and conjunction through the symbolic " $\cdot$ ", we arrive at the following definition of provenance as sum of products.

Definition 2.4. *Let $A(Q, D)$ be the set of assignments for a CQ Q and a database instance D. We define the* value-level provenance *of Q w.r.t. D as*

$$\sum_{\alpha \in A(Q,D)} \Pi_{\{x_i, a_i | \alpha(x_i) = a_i\}}(x_i, a_i)$$

.

Rel. *org*

| oid | oname |
|---|---|
| 1 | UPENN |
| 2 | TAU |

Rel. *author*

| aid | aname | oid |
|---|---|---|
| 3 | Susan D. | 1 |
| 4 | Tova M. | 2 |
| 5 | Slava N. | 2 |

Rel. *pub*

| wid | cid | ptitle | pyear |
|---|---|---|---|
| 6 | 10 | "OASSIS..." | 2014 |
| 7 | 10 | "A sample..." | 2014 |
| 8 | 11 | "Monitoring..." | 2007 |
| 9 | 11 | "Querying..." | 2006 |

Rel. *writes*

| aid | wid |
|---|---|
| 4 | 6 |
| 3 | 6 |
| 5 | 6 |
| 4 | 7 |
| 4 | 8 |
| 4 | 9 |

Rel. *conf*

| cid | cname |
|---|---|
| 10 | SIGMOD |
| 11 | VLDB |

Rel. *domainConf*

| cid | did |
|---|---|
| 10 | 18 |
| 11 | 18 |

Rel. *domain*

| did | name |
|---|---|
| 18 | Databases |

**Figure 6:** DB Instance

Example 2.5. *Re-consider our running example query and consider the database in Figure 6. The value-level provenance is shown in Figure 5. Each of the 6 summands stands for a different assignment (i.e. an alternative reason for the tuple to appear in the result). Assignments are represented as multiplication of pairs of the form $(var, val)$ so that var is assigned val in the particular assignment. We only show here variables to which a query word was mapped; these will be the relevant variables for formulating the answer.*

By composing the dependency-to-query-mapping from the NL query's dependency tree to query variables, and the assignments of query variables to values from the database, we associate different parts of the NL query with values. We will use this composition of mappings throughout the paper as a means of assembling the NL answer to the NL query.

Example 2.6. *Continuing our running example, consider the assignment represented by the first monomial of Figure 5. Further reconsider Figure 4a, and now note that each node is associated with a pair $(var, val)$ of the variable to which the node was mapped, and the value that this variable was assigned in this particular assignment. For instance, the node "organization" was mapped to the variable oname which was assigned the value "TAU".*

# 3. FIRST STEP: A SINGLE ASSIGNMENT

We start describing our transformation of provenance to NL for a single assignment. The solution will serve as the basis for the general case of multiple assignments.

## 3.1 Basic Solution

We follow the structure of the NL query dependency tree and generate an answer tree with the same structure by replacing/modifying the words in the question with the values from the result and provenance that were mapped using the dependency-to-query-mapping and the assignment. Yet, note that simply replacing the values does not always result in a coherent sentence, as shown in the following example.

EXAMPLE 3.1. *Re-consider the dependency tree depicted in Figure 4a. If we were to replace the value in the organization node to the value "TAU" mapped to it, the word "organization" will not appear in the answer although it is needed to produce the coherent answer depicted in Figure 2. Without this word, it is unclear how to deduce the information about the connection between "Tova M." and "TAU".*

We next account for these difficulties and exemplify our approach that outputs the dependency tree of a detailed answer; We do so by augmenting the query dependency tree into an answer tree. we will further translate this tree to an NL sentence.

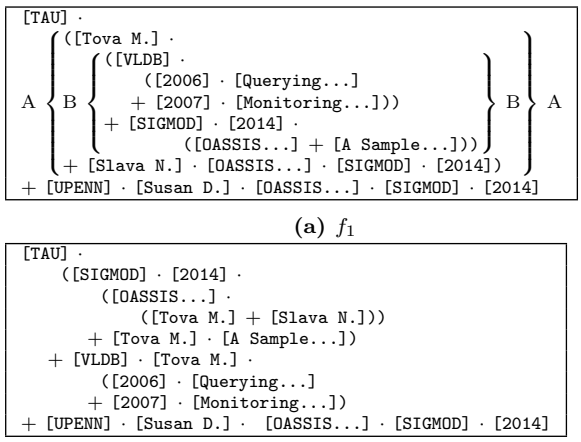
Recall that the dependency tree of the NL query follows one of the abstract forms in Figure 3. We distinguish between two cases based on nodes whose *REL* (relationship with parent node) is *modifier*; in the first case, the clause begins with a verb modifier (*e.g.*, the node "published" in Fig. 4a is a verb modifier) and in the second, the clause begins with a non-verb modifier (*e.g.*, the node "of" in Fig. 4a is a non-verb modifier). In short, the children of verb modifier nodes are replaced with the value mapped to them while the children of non-verb modifier nodes stay as part of the tree and the value mapped to them is added to the tree.

EXAMPLE 3.2. *Re-consider Figure 4a, and note the mappings from the nodes to the variables and values as reflected in the boxes next to the nodes. To generate an answer, we follow the NL query structure, "plugging-in" mapped database values. We start with "organization", which is the first node to be considered. Observe that "organization" has the child "of" which is a non-verb modifier, so we add "TAU" as its child. On the other hand, the node "authors" has the child "published" which is a verb modifier, so we replace "authors" with the value "Tova M.", mapped to it. Another case is the handling of the nodes "after" and "in" which are modifiers as well. These nodes refer to times and locations, hence we replace the subtree rooted at these nodes with the node mapped to their child (in the case of "after" it is "2014" and in the case of "in" it is "SIGMOD") and attach the node "in" as the parent of the node, in both cases as it is the suitable word for equality for years and locations.*

So far we have augmented the NL query dependency tree to obtain the dependency tree of the answer. The last step is to translate this tree to a sentence. To this end, we recall that the original query, in the form of a sentence, was translated by `NaLIR` to the NL query dependency tree. To translate the dependency tree to a sentence, we essentially "revert" this process, further using the mapping of NL query dependency tree nodes to (sets of) nodes of the answer.

# 4. THE GENERAL CASE

In general, as illustrated in Section 2, the provenance may include multiple assignments. We next generalize the construction to account for this. Note that a naïve solution in this respect is to generate a sentence for each individual assignment and concatenate the resulting sentences. However, already for the small-scale example presented here, this would result in a long and unreadable answer (recall Figure 5 consisting of six assignments). Instead, we propose two solutions: the first based on the idea of provenance factorization [18, 4], and the second leveraging factorization to provide a summarized form.




**(a)** $f_1$




**(b)** $f_2$

**Figure 7:** Provenance Factorizations

## 4.1 NL-Oriented Factorization

We start by defining the notion of factorization in a standard way (see *e.g.* [18, 8]).

DEFINITION 4.1. *Let $P$ be a provenance expression. We say that an expression $f$ is a factorization of $P$ if $f$ may be obtained from $P$ through (repeated) use of some of the following axioms: distributivity of summation over multiplication, associativity and commutativity of both summation and multiplication.*

EXAMPLE 4.2. *Re-consider the provenance expression in Figure 5. Two possible factorizations are shown in Figure 7, keeping only the values and omitting the variable names for brevity (ignore the A,B brackets for now). In both cases, the idea is to avoid repetitions in the provenance expression, by taking out a common factor that appears in multiple summands. Different choices of which common factor to take out lead to different factorizations.*

How do we measure whether a possible factorization is suitable/preferable to others? Standard desiderata [18, 8] are that it should be short or that the maximal number of appearances of an atom is minimal. On the other hand, we factorize here as a step towards generating an NL answer; to this end, it will be highly useful if the *(partial) order of nesting of value annotations in the factorization is consistent the (partial) order of corresponding words in the NL query.* We will next formalize this intuition as a constraint over factorizations. We start by defining a partial order on nodes in a dependency tree:

DEFINITION 4.3. *Given an dependency tree $T$, we define $\leq_T$ as the descendant partial order of nodes in $T$: for each two nodes, $x, y \in V(T)$, we say that $x \leq_T y$ if $x$ is a descendant of $y$ in $T$.*

EXAMPLE 4.4. *In our running example (Figure 4a) it holds in particular that $authors \leq organization$, $2005 \leq authors$, $conferences \leq authors$ and $papers \leq authors$, but $papers$, $2005$ and $conferences$ are incomparable.*

Next we define a partial order over elements of a factorization, intuitively based on their nesting depth. To this end, we first consider the *circuit form* [3] of a given factorization:

Example 4.5. *Consider the partial circuit of $f_1$ in Figure 8. The root, $\cdot$, has two children; the left child is the leaf "TAU" and the right is a $+$ child whose subtree includes the part that is "deeper" than "TAU".*

Given a factorization $f$ and an element $n$ in it, we denote by $level_f(n)$ the distance of the node $n$ from the root of the circuit induced by $f$ multiplied by $(-1)$. Intuitively, $level_f(n)$ is bigger for a node $n$ closer to the circuit root.

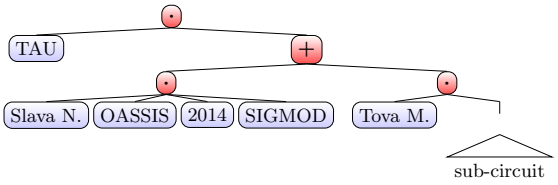



**Figure 8:** Sub-Circuit of $f_1$

Our goal here is to define the correspondence between the level of each node in the circuit and the level of its "source" node in the NL query's dependency tree (note that each node in the query corresponds to possibly many nodes in the circuit: all values assigned to the variable in the different assignments). In the following definition we will omit the database instance for brevity and denote the provenance obtained for a query with dependency tree $T$ by $prov_T$. Recall that dependency-to-query-mapping maps the nodes of the dependency tree to the query variables and the assignment maps these variables to values from the database.

Definition 4.6. *Let $T$ be a query dependency tree, let $prov_T$ be a provenance expression, let $f$ be a factorization of $prov_T$, let $\tau$ be a dependency-to-query-mapping and let $\{\alpha_1, ...\alpha_n\}$ be the set of assignments to the query. For each two nodes $x, y$ in $T$ we say that $x \leq_f y$ if $\forall i \in [n] : level_f(\alpha_i(\tau(x))) \leq level_f(\alpha_i(\tau(y)))$.*

*We say that $f$ is $T$-compatible if each pair of nodes $x \neq y \in V(T)$ that satisfy $x \leq_T y$ also satisfy that $x \leq_f y$.*

Essentially, $T$-compatibility means that the partial order of nesting between values, for each individual assignment, must be consistent the partial order defined by the structure of the question. Note that the compatibility requirement imposes constraints on the factorization, but it is in general far from dictating the factorization, since the order $x \leq_T y$ is only partial – and there is no constraint on the order of each two provenance nodes whose "origins" in the query are unordered. Among the $T$-compatible factorizations, we will prefer shorter ones.

Definition 4.7. *Let $T$ be an NL query dependency tree and let $prov_T$ be a provenance expression for the answer. We say that a factorization $f$ of $prov_T$ is* optimal *if $f$ is $T$-compatible and there is no $T$-compatible factorization $f'$ of $prov_T$ such that $|f'| < |f|$ ($|f|$ is the length of $f$).*

The following example shows that the $T$-compatibility constraint still allows much freedom in constructing the factorization. In particular, different choices can (and sometimes should, to achieve minimal size) be made for different subexpressions, including ones leading to different answers and ones leading to the same answer through different assignments.

Example 4.8. *Recall the partial order $\leq_T$ imposed by our running example query, shown in part in Example 4.4. It implies that in every compatible factorization, the organization name must reside at the highest level, and indeed TAU was "pulled out" first in Figure 8; similarly the author name must be pulled out next. In contrast, since the query nodes corresponding to title, year and conference name are unordered, we may,* within a single factorization*, factor out e.g. the year in one part of the factorization and the conference name in another one. As an example, Tova M. has two papers published in VLDB ("Querying..." and "Monitoring") so factorizing based on VLDB would be the best choice for that part. On the other hand, suppose that Slava N. had two paper published in* 2014*; then we could factorize them based on* 2014*. The factorization could, in that case, look like the following (where the parts taken out for Tova and Slava are shown in bold):*

```
[TAU] ·
([Tova M.] ·
([VLDB] ·
    ([2006] · [Querying...]
  + [2007] · [Monitoring...]))
+ [SIGMOD] · [2014] ·
        ([OASSIS...] + [A Sample...]))
+ ([Slava N.] ·
  ([2014] ·
    ([SIGMOD] · [OASSIS...]
   + [VLDB] · [Ontology...])))
```

The following example shows that in some cases, requiring compatibility can come at the cost of compactness.

Example 4.9. *Consider the query tree $T$ depicted in Figure 4a and the factorizations $prov_T$ (the identity factorization) depicted in Figure 5, $f_1$, $f_2$ presented in Figure 7. $prov_T$ is of length* 30 *and is* 5*-readable, i.e., the maximal number of appearances of a single variable is* 5 *(see [8]). $f_1$ is of length* 20*, while the length of $f_2$ is only* 19*. In addition, both $f_1$ and $f_2$ are* 3*-readable. Based on those measurements $f_2$ seems to be the best factorization, yet $f_1$ is $T$-compatible with the question and $f_2$ is not. For example, conferences $\leq_T$ authors but "SIGMOD" appears higher than "Tova M." in $f_2$. Choosing a $T$-compatible factorization in $f_1$ will lead (as shown below) to an answer whose structure resembles that of the question, and thus translates to a more coherent and fitting NL answer.*

Note that the identity factorization is always $T$-compatible, so we are guaranteed at least one optimal factorization (but it is not necessarily unique). We next study the problem of computing such a factorization.

## 4.2 Computing Factorizations

Recall that our notion of compatibility restricts the factorizations so that their structure resembles that of the question. Without this constraint, finding shortest factorizations is coNP-hard in the size of the provenance (*i.e.* a boolean expression) [13]. The compatibility constraint does not reduce the complexity since it only restricts choices relevant to part of the expression, while allowing freedom for arbitrarily many other elements of the provenance. Also recall (Example 4.8) that the choice of which element to "pull-out" needs in general to be done separately for each part of the provenance so as to optimize its size (which is the reason for the hardness in [13] as well). In general:

PROPOSITION 4.10. *Given a dependency tree $T$, a provenance expression $prov_T$ and an integer $k$, deciding whether there exists a $T$-compatible factorization of $prov_T$ of size $\leq k$ is coNP-hard.*

**Greedy Algorithm.** Despite the above result, the constraint of compatibility does help in practice, in that we can avoid examining choices that violate it. For other choices, we devise a simple algorithm that chooses greedily among them. More concretely, the input to Algorithm 1 is the query tree $T_Q$ (with its partial order $\leq_{T_Q}$), and the provenance $prov_{T_Q}$. The algorithm output is a $T_Q$-compatible factorization $f$. Starting from *prov*, the progress of the algorithm is made in steps, where at each step, the algorithm traverses the circuit induced by *prov* in a BFS manner from top to bottom and takes out a variable that would lead to a minimal expression out of the valid options that keep the current factorization $T$-compatible. Naturally, the algorithm does not guarantee an optimal factorization (in terms of length), but performs well in practice.

In more detail, we start by choosing the largest nodes according to $\leq_{T_Q}$ which have not been processed yet (Line 2). Afterwards, we sort the corresponding variables in a greedy manner based on the number of appearances of each variable in the expression using the procedure $sortByFrequentVars$ (Line 3). In Lines 4–5, we iterate over the sorted variables and extract them from their sub-expressions. This is done while preserving the $\leq_{T_Q}$ order with the larger nodes, thus ensuring that the factorization will remain $T_Q$-compatible. We then add all the newly processed nodes to the set *Processed* which contains all nodes that have already been processed (Line 6). Lastly, we check whether there are no more nodes to be processed, *i*.e., if the set *Processed* includes all the nodes of $T_Q$ (denoted $V(T_Q)$, see the condition in Line 7). If the answer is "yes", we return the factorization. Otherwise, we make a recursive call. In each such call, the set *Processed* becomes larger until the condition in Line 7 holds.

**Algorithm 1:** GreedyFactorization

**input** : $T_Q$ - the query tree, $\leq_{T_Q}$ - the query partial order, *prov* - the provenance, $\tau, \alpha$ - dependency-to-query-mapping and assignment from nodes in $T_Q$ to provenance variables, *Processed* - subset of nodes from $V(T_Q)$ which were already processed (initially, $\emptyset$)

**output**: $f$ - $T_Q$-compatible factorization of $prov_{T_Q}$

1 $f \leftarrow prov$;
2 $Frontier \leftarrow \{x \in V(T_Q) | \forall (y \in V(T_Q) \setminus Processed)\ s.t.\ x \not\leq_{T_Q} y\}$;
3 $vars \leftarrow sortByFrequentVars(\{\alpha(\tau(x)) | x \in Frontier\}, f)$;
4 **foreach** $var \in vars$ **do**
5  Take out *var* from sub-expressions in $f$ not including variables from $\{x | \exists y \in Processed : x = \alpha(\tau(y))\}$;
6 $Processed \leftarrow Processed \cup Frontier$;
7 **if** $|Processed| = |V(T_Q)|$ **then**
8  **return** $f$;
9 **else**
10  **return** $GreedyFactorization(T_Q, f, \tau, \alpha, Processed)$;

EXAMPLE 4.11. *Consider the query tree $T_Q$ depicted in Figure 4a, and provenance prov in Figure 5. As explained above, the largest node according to $\leq_{T_Q}$ is organization, hence "TAU" will be taken out from the brackets multiplying all summands that contain it. Afterwards, the next node according to the order relation will be author, therefore we group by author, taking out "Tova M.", "Slava N." etc. The following choice (between conference, year and paper name) is then done greedily* for each author*, based on its number of occurrences. For instance, $VLDB$ appears twice for $Tova.M.$ whereas each paper title and year appears only once; so it will be pulled out. The polynomial $[SlavaN.] \cdot [OASSIS...] \cdot [SIGMOD] \cdot [2014]$ will remain unfactorized as all values appear once. Eventually, the algorithm will return the factorization $f_1$ depicted in Figure 7, which is $T_Q$-compatible and much shorter than the initial provenance expression.*

PROPOSITION 4.12. *Let $f$ be the output of Algorithm 1 for the input dependency tree $T_Q$, then $f$ is $T_Q$-compatible.*

***Complexity.*** Denote the provenance size by $n$. The algorithm complexity is $O(n^2 \cdot \log n)$: at each recursive call, we sort all nodes in $O(n \cdot \log n)$ (Line 3) and the we handle (in $Frontier$) at least one node (in the case of a chain graph) or more. Hence, in the worst case we would have $n$ recursive calls, each one costing $O(n \cdot \log n)$.

## 4.3 Factorization to Answer Tree

The final step is to turn the obtained factorization into an NL answer. Similarly to the case of a single assignment (Section 3), we leverage the mappings and assignments to convert the query dependency tree into an answer tree that reflects the factorization. Intuitively, we follow the structure of a single answer, replacing each node there by either a single node, standing for a single word of the factorized expression, or by subtree, standing for some brackets (subcircuit) in the factorized expression.

EXAMPLE 4.13. *Consider the factorization $f_1$ depicted in Figure 7, and the structure of single assignment answer depicted in Figure 4b which was built based on an answer tree for a single assignment. Given this input, we will generate an answer tree corresponding to the following sentence:*

```
TAU is the organization of
    Tova M. who published
        in VLDB
           'Querying...' in 2006 and
           'Monitoring...' in 2007
        and in SIGMOD in 2014
            'OASSIS...' and  'A sample...'
    and Slava N. who published
        'OASSIS...' in SIGMOD in 2014.
UPENN is the organization of Susan D. who published
'OASSIS...' in SIGMOD in 2014.
```

*Note that the query has two results: "TAU" and "UPENN". "UPENN" was produced with a single assignment, but there are five different assignments producing "TAU". Focusing on the factorization part of the result "TAU", notice that the authors were pulled out first, then the conferences, and then the years and papers, so this will be reflected in the factorized answer tree. For example, we replace the node authors with the values from the factorization that correspond to this word, i.e., Tova M. and Slava N. The answer tree can also be changed based on the hierarchy of the factorization. For instance, although the node paper is closer to the root of the tree then the nodes year and conference in the original answer tree, the order of these nodes in the new answer*

*tree will be reversed since $f_1$ extracted the values "VLDB", "SIGMOD" and "2014".*

**Why require compatibility?** We conclude this part of the paper by revisiting our decision to require compatible factorizations, highlighting difficulties in generating NL answers using non-compatible factorizations.

EXAMPLE 4.14. *Consider factorization $f_2$ from Figure 7. "TAU" should be at the beginning of the sentence and followed by the conference names "SIGMOD" and "VLDB". The second and third layers of $f_2$ are composed of author names ("Tova M.", "Slava N."), paper titles ("OASSIS", "A sample...", "Monitoring...") and publication years (*2007, 2014*). Changing the original order of the words such that the conference name "SIGMOD" and the publication year "2014" will appear before "Tova M." breaks the sentence structure in a sense. It is unclear how to algorithmically translate this factorization into an NL answer, since we need to patch the broken structure by adding connecting phrases. One hypothetical option of patching $f_2$ and transforming it into an NL answer is depicted below. The bold parts of the sentence are not part of the factorization and it is not clear how to generate and incorporate them into the sentence algorithmically. Even if we could do so, it appears that the resulting sentence would be quite convoluted:*

```
TAU is the organization of authors who published in
    SIGMOD 2014
        'OASSIS...' which was published by
            Tova M. and Slava N.
        and Tova M. published 'A sample...'
    and Tova M. published in VLDB
        'Querying...' in 2014
        and 'Monitoring...' in 2007.
UPENN is the organization of Susan D. who published
'OASSIS...' in SIGMOD in 2014
```

*Observe that the resulting sentence is much less clear than the one obtained through our approach, even though it was obtained from a shorter factorization $f_2$; the intuitive reason is that since $f_2$ is not $T$-compatible, it does not admit a structure that is similar to that of the question, thus is not guaranteed to admit a structure that is coherent in Natural Language. Interestingly, the sentence we would obtain in such a way also has an edit distance from the question [9] that is shorter than that of our answer, demonstrating that edit distance is not an adequate measure here.*

## 4.4 From Factorizations to Summarizations

When there are many assignments and/or the assignments involve multiple distinct values, even an optimal factorized representation may be too long and convoluted for users to follow.

EXAMPLE 4.15. *Reconsider Example 4.13; if there are many authors from TAU then even the compact representation of the result could be very long. In such cases we need to summarize the provenance in some way that will preserve the "essence" of all assignments without actually specifying them, for instance by providing only the number of authors/-papers for each institution.*

To this end, we employ *summarization*, as follows. First, we note that a key to summarization is understanding which parts of the provenance may be grouped together. For that, we use again the mapping from nodes to query variables:

```
(A) [TAU] · Size([Tova M.],[Slava N.]) · Size([VLDB],[SIGMOD]) ·
      Size([Querying...],[Monitoring...],
       [OASSIS...],[A Sample...]) · Range([2006],[2007],[2014])
(B) [TAU]·(
     [Tova M.] ·
       Size([VLDB],[SIGMOD]) ·
       Size([Querying...],[Monitoring...],
        [OASSIS...],[A Sample...]) · Range([2006],[2007],[2014])
     [Slava N.] · [OASSIS...] · [SIGMOD] · [2014])
```

**Figure 9:** Summarized Factorizations

```
(A) TAU is the organization of 2 authors who published
4 papers in 2 conferences in 2006 - 2014.
(B) TAU is the organization of Tova M. who published
4 papers in 2 conferences in 2006 - 2014 and Slava N.
who published 'OASSIS...' in SIGMOD in 2014.
```

**Figure 10:** Summarized Sentences

we say that two nodes are of the same *type* if both were mapped to the same query variable. Now, let $n$ be a node in the circuit form of a given factorization $f$. A summarization of the sub-circuit of $n$ is obtained in two steps. First, we group the descendants of $n$ according to their type. Then, we summarize each group separately. The latter is done in our implementation simply by either counting the number of distinct values in the group or by computing their range if the values are numeric. In general, one can easily adapt the solution to apply additional user-defined "summarization functions" such as "greater / smaller than X" (for numerical values) or "in continent Y" for countries.

EXAMPLE 4.16. *Re-consider the factorization $f_1$ from Figure 7. We can summarize it in multiple levels: the highest level of authors (summarization "A"), or the level of papers for each particular author (summarization "B"), or the level of conferences, etc. Note that if we choose to summarize at some level, we must summarize its entire sub-circuit (e.g. if we summarize for Tova. M. at the level of conferences, we cannot specify the papers titles and publication years).*

*Figure 9 presents the summarizations of sub-trees for the "TAU" answer, where "size" is a summarization operator that counts the number of distinct values and "range" is an operator over numeric values, summarizing them as their range. The summarized factorizations are further converted to NL sentences which are shown in Figure 10. Summarizing at a higher level results in a shorter but less detailed summarization.*

# 5. RELATED WORK

Multiple lines of work (*e.g.* [2, 17, 21]) have proposed NL interfaces to formulate database queries, and additional works [10] have focused on presenting the *answers* in NL, typically basing their translation on the schema of the output relation. Among these, works such as [2, 17] also harness the dependency tree in order to make the translation form NL to SQL by employing *mappings* from the NL query to formal terms. To our knowledge, no previous work has focused on formulating the *provenance* of output tuples in NL. This requires fundamentally different techniques (*e.g.* that of factorization and summarization, building the sentence based on the input question structure, etc.) and leads to answers of much greater detail.

The tracking, storage and presentation of provenance have been the subject of extensive research (see *e.g.* [12, 14, 6]) while the field of provenance applications has also been

**Table 1:** Sample use-cases and results

| Query | Single Assignment | Multiple Assignments - Summarized |
|---|---|---|
| Return the homepage of SIGMOD | http://www.sigmod2011.org/ is the homepage of SIGMOD | |
| Return the authors who published papers in SIGMOD before 2015 and after 2005 | Tova M. published "Auto-completion..." in SIGMOD in 2012 | Tova M. published 10 papers in SIGMOD in 2006-2014 |
| Return the authors from TAU who published papers in VLDB | Tova M. from TAU published "XML Repository..." in VLDB | Tova M. from TAU published 11 papers in VLDB |
| Return the authors who published papers in database conferences | Tova M. "published Auto-completion..." in SIGMOD | Tova M. published 96 papers in 18 conferences |
| Return the organization of authors who published papers in database conferences after 2005 | TAU is the organization of Tova M. who published 'OASSIS...' in SIGMOD in 2014 | TAU is the organization of 43 authors who published 170 papers in 31 conferences in 2006 - 2015 |

broadly studied (*e.g.* [7, 19]). A longstanding challenge in this context is the complexity of provenance expressions, leading to difficulties in presenting them in a user comprehensible manner. Approaches in this respect include showing the provenance in a graph form (see *e.g.* [20, 6]), allowing user control over the level of granularity ("zooming" in and out [5]), or otherwise presenting different ways of provenance visualization [14]. Other works have studied allowing users to query the provenance (*e.g.* [16, 15]) or to a-priori request that only parts of the provenance are tracked (see for example [7, 11]). Importantly, provenance factorization and summarization have been studied (*e.g.*, [1, 18, 4]) as means for compact representation of the provenance. Usually, the solutions proposed in these works aim at reducing the size of the provenance but naturally do not account for its presentation in NL; we have highlighted the different considerations in context of factorization/summarization in our setting.

# 6. CONCLUSION AND LIMITATIONS

We have studied in this paper, for the first time to our knowledge, provenance for NL queries. We have devised means for presenting the provenance information again in Natural Language, in factorized or summarized form.

There are two main limitations to our work. First, a part of our solution was designed to fit `NaLIR`, and will need to be replaced if a different NL query engine is used. Specifically, the "sentence generation" module will need to be adapted to the way the query engine transforms NL queries into formal ones; our notions of factorization and summarization are expected to be easier to adapt to a different engine. Second, our solution is limited to Conjunctive Queries. One of the important challenges in supporting NL provenance for further constructs such as union and aggregates is the need to construct a concise presentation of the provenance in NL.

*Acknowledgments.* This research was partially supported by the Israeli Science Foundation (ISF, grant No. 1636/13), and by ICRC - The Blavatnik Interdisciplinary Cyber Research Center. The contribution of Amir Gilad is part of a Ph.D. thesis research conducted at Tel Aviv University.

# 7. REFERENCES

[1] E. Ainy, P. Bourhis, S. B. Davidson, D. Deutch, and T. Milo. Approximated summarization of data provenance. In *CIKM*, pages 483–492, 2015.
[2] Y. Amsterdamer, A. Kukliansky, and T. Milo. A natural language interface for querying general and individual knowledge. *VLDB*, pages 1430–1441, 2015.
[3] P. Brgisser, M. Clausen, and M. A. Shokrollahi. *Algebraic Complexity Theory*. Springer Publishing Company, Incorporated, 2010.
[4] A. P. Chapman, H. V. Jagadish, and P. Ramanan. Efficient provenance storage. In *SIGMOD*, pages 993–1006, 2008.
[5] S. Cohen-Boulakia, O. Biton, S. Cohen, and S. Davidson. Addressing the provenance challenge using zoom. *Concurr. Comput. : Pract. Exper.*, pages 497–506, 2008.
[6] S. B. Davidson and J. Freire. Provenance and scientific workflows: challenges and opportunities. In *SIGMOD*, pages 1345–1350, 2008.
[7] D. Deutch, A. Gilad, and Y. Moskovitch. Selective provenance for datalog programs using top-k queries. *PVLDB*, pages 1394–1405, 2015.
[8] K. Elbassioni, K. Makino, and I. Rauf. On the readability of monotone boolean formulae. *JoCO*, pages 293–304, 2011.
[9] M. Emms. Variants of tree similarity in a question answering task. In *Proceedings of the Workshop on Linguistic Distances*, pages 100–108, 2006.
[10] E. Franconi, C. Gardent, X. I. Juarez-Castro, and L. Perez-Beltrachini. Quelo Natural Language Interface: Generating queries and answer descriptions. In *Natural Language Interfaces for Web of Data*, 2014.
[11] B. Glavic. Big data provenance: Challenges and implications for benchmarking. In *Specifying Big Data Benchmarks - First Workshop, WBDB*, pages 72–80, 2012.
[12] T. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *PODS*, pages 31–40, 2007.
[13] E. Hemaspaandra and H. Schnoor. Minimization for generalized boolean formulas. In *IJCAI*, pages 566–571, 2011.
[14] M. Herschel and M. Hlawatsch. Provenance: On and behind the screens. In *SIGMOD*, pages 2213–2217, 2016.
[15] Z. G. Ives, A. Haeberlen, T. Feng, and W. Gatterbauer. Querying provenance for ranking and recommending. In *TaPP*, pages 9–9, 2012.
[16] G. Karvounarakis, Z. G. Ives, and V. Tannen. Querying data provenance. In *SIGMOD*, pages 951–962, 2010.
[17] F. Li and H. V. Jagadish. Constructing an interactive natural language interface for relational databases. *Proc. VLDB Endow.*, pages 73–84, 2014.
[18] D. Olteanu and J. Závodný. Factorised representations of query results: Size bounds and readability. In *ICDT*, pages 285–298, 2012.
[19] S. Roy and D. Suciu. A formal approach to finding explanations for database queries. In *SIGMOD*, pages 1579–1590, 2014.
[20] Y. L. Simmhan, B. Plale, and D. Gannon. Karma2: Provenance management for data-driven workflows. *Int. J. Web Service Res.*, pages 1–22, 2008.
[21] D. Song, F. Schilder, C. Smiley, C. Brew, T. Zielund, H. Bretz, R. Martin, C. Dale, J. Duprey, T. Miller, and J. Harrison. TR discover: A natural language interface for querying and analyzing interlinked datasets. In *ISWC*, pages 21–37, 2015.