Bulletin of the Technical Committee on

Data Engineering

September 2014 Vol. 37 No. 3

IEEE Computer Society

Letters

Special Issue on Databases, Declarative Systems and Machine Learning

Conference and Journal Notices

The TC on Data Engineering

Membership in the TC on Data Engineering is open to all current members of the IEEE Computer Society who are interested in database systems. The TCDE web page is http://tab.computer.org/tcde/index.html.

The Data Engineering Bulletin

The Bulletin of the Technical Committee on Data Engineering is published quarterly and is distributed to all TC members. Its scope includes the design, implementation, modelling, theory and application of database systems and their technology.

Letters, conference information, and news should be sent to the Editor-in-Chief. Papers for each issue are solicited by and should be sent to the Associate Editor responsible for the issue.

Opinions expressed in contributions are those of the authors and do not necessarily reflect the positions of the TC on Data Engineering, the IEEE Computer Society, or the authors' organizations.

The Data Engineering Bulletin web site is at http://tab.computer.org/tcde/bull_about.html.

Letter from the Editor-in-Chief

TCDE Chair Election

Voting is IMPORTANT. Every two years, the Technical Committee on Data Engineering has an election for a new TCDE chair. The chair is the person who is empowered and entrusted with the authority to appoint members of the TCDE Executive Committee. The Executive Committee has full authority in managing the Technical Committee. So when you vote, you are chosing someone who is critical to the on-going operation of the Technical Committee. Elections have consequences.

Page two of this issue has a letter from the nominating committee describing their work and naming the candidates. Also on page two, Brookes Little from the IEEE Computer Society describes how you can vote. This is followed on pages three and four with candidate statements and biographies. Brookes describes how you can vote, but only you can exercise that function. Please do.

The Current Issue

Machine learning (ML) is a huge deal, driven by the lure of finding the proverbial needle in the haystack insight that is transformative. Also driven by the fear that someone else will find the needle first. So this is a hugely competitive space, with implications for business, science, education, indeed a vast array of areas. This issue should be of interest to any number of database technologists, be they practitioners or researchers, data scientists or data platform specialists.

This is more than simply taking a look at Map/Reduce. That is pure infrastructure, and now only part of the platform space that includes distributed databases as well as more specialized infrastructures. And platform work only captures part of what is going on. Like OLAP, one needs more than just a platform, one needs to bring machine learning methods into play. Inferencing, probability, model building, categorizing, and more can be usefully integrated into ML systems. While data intensive ML related work has gone on for a number of years, this is nonetheless a young area with enormous potential for high impact work that may span the better part of many careers.

Thus, ML is a great topic for the Data Engineering Bulletin. Chris Jermaine is the editor for this introductory issue on ML, but I am sure there will be more ML Bulletin issues in the future. The current issue spans platforms, models, scaling, ML approaches, and more. Chris has assembled a diverse collection of articles in this area that can function as that can function as an introduction to the area, and the start of a journey to an in-depth understanding of ML technology and its uses. I want to thank Chris for this fine result, and for some additional sweat in dealing with Bulletin formatting difficulties.

David Lomet
Microsoft Corporation

Message from the TCDE Chair Nominating Committee

The Chair of the IEEE Computer Society Technical Committee on Data Engineering (TCDE) is elected for a two-year period. The mandate of the current Chair, Kyu-Young Whang, is coming to an end and it is time to elect a Chair for the next two years.

The TCDE Chair Nominating Committee, consisting of Amr El Abaddi, Paul Larson, and David Lomet, has nominated two candidates for the position as Chair of TCDE: Erich Neuhold and Xiaofang Zhou. The Committee regards both candidates as highly qualified to fill the office of Chair of the Technical Committee, and urges members to cast their votes.

More information about TCDE can be found at http://tab.computer.org/tcde/ as well as in the accompanying election announcement below from Brookes Little of the Computer Society.

David Lomet, Paul Larson and Amr El Abbadi
Microsoft, Microsoft, and UC Santa Barbara

Election for Chair of IEEE Computer Society TC on Data Engineering

The 2014 TCDE Chair election is now open. The Chair's term will run fromJanuary 1, 2015 through December 31, 2016.

The poll will close November 3, 2014, at5:00pmPacific time.

Before entering the poll, please take a few moments to review candidate Biosketches and Position Statements, which can be found here: http://www.computer.org/portal/web/tandc/tcde

Please note: Only Computer Society (CS) Members who are also TCDE Members can vote. **To cast your vote, you will need your IEEE CS Member number. To obtain a misplaced number, please visit:
https://supportcenter.ieee.org/app/answers/detail/a_id/353/session/L3RpbWUvMTM3NTc0NDYxMi9zaWQvdXIxcVAxeGw%3D

To ensure your TC membership is valid and up to date, please log in to your IEEE Web Account and check your membership status: https://www.ieee.org/profile/public/login/publiclogin.html

You may vote only one time.

VOTE here: https://www.surveymonkey.com/s/HHMDH8K
If you have trouble voting, please contact T&C Senior Program Specialist, Brookes Little (eblittle@computer.org).

Thank you for your participation in this election!

Brookes Little
IEEE Computer Society

TCDE Chair Candidate Erich Neuhold

Biography

Erich J. Neuhold is currently Honorary (Adjunct) Professor for Computer Science at the University of Vienna and associated with the Fakultaet fuer Informatik. Here he is involved with structured and unstructured multimedia Databases and Web-Information Systems and questions of Semantic enrichment for effective Information Interoperability and Mining. He is also investigating issues of interoperability, security and privacy as they arise in the WEB 2.0 environment and all the social communities whether private or professional.

Until April 2005 he was Professor of Computer Science at the University of Technology in Darmstadt and Director of the Fraunhofer Institute for Integrated Publication and Information Systems (IPSI) in Darmstadt, Germany an institution of about 120 persons involved in all aspects of Information systems and Internet Services. A number of spin-offs were started during his leadership to exploit the results of IPSIs research.

He is or has been on the Steering Committees of the main Data Base Conferences VLDB, ICDE and also the main Digital Library Conferences JCDL, ECDL and ICADL. He is or has been a member of various editorial boards and currently serves as one of the Chairs of the IJDL Editorial Board. He is a Fellow of IEEE, USA and of the Gesellschaft fuer Informatik, Germany.

He has published four books and about 200 papers.

Position statement

If elected as the new TCDE Chair, I basically will continue the activities of the team established by the current Chair Kyu-Young Whang. I feel he and his team have done an excellent job and I will keep all of the members, if they so agree. Our membership has increased significantly and is still growing. Special attention should also be given to gain more members from organizations that are users of our technologies. Their input would make our conferences of higher interest to such persons and increase attendance. Our flagship conference series ICDE is doing very well and our Bulletin (edited by David Lomet) has a very high worldwide reputation. We also have two Working Groups, (1) Self-Managing DB Working Group and (2) Cloud Data Management Working Group, that are contributing significantly to the scope of our TC. Here I would make an effort to establish even more Working Groups, as they are flexible vehicles to cover newly arising important subjects (e.g. Big Data, Data Security and Privacy). It is interesting to note that that in the two established groups the processing aspect of data is of significance and this is even more so in a Cloud or Secure Data environment. A trust in data stores does not come from the data but from what is done with them. Here, I believe, our community can still make significant contributions to (re)establish trust in the Big data collections around, whether in company/government collections or in public/private clouds.

Overall I would say TCDE is sailing very well and I would not rock such a boat.

Erich Neuhold
Vienna University
Vienna, Austria

TCDE Chair Candidate Xiaofang Zhou

Biography

Xiaofang Zhou is a Professor of Computer Science at the University of Queensland, leading its Data and Knowledge Engineering research group. He received his BSc and MSc in Computer Science from Nanjing University, China, and PhD in Computer Science from the University of Queensland. Before joining UQ in 1999, he worked as a researcher in CSIRO. His research focus is to find effective and efficient solutions for managing, integrating and analyzing very large amount of complex data for business, scientific and personal applications. He has been working in the area of spatial and multimedia databases, data quality, high performance query processing, Web information systems and bioinformatics, co-authored over 250 research papers with many published in top journals and conferences such as ICDE, SIGMOD, VLDB, The VLDB Journal, and various ACM and IEEE Transactions. He served the research community in various capacities including being Director of ARC Research Network in Enterprise Information Infrastructure, a major national research collaboration initiative in Australia, an Associate Editor of The VLDB Journal, IEEE Transactions on Data and Knowledge Engineering, IEEE Transactions on Cloud Computing, WWW Journal, Distributed and Parallel Databases, and IEEE Data Engineering Bulletin, a PC co-chair of ICDE, DASFAA, WISE, CoopIS and ADC, a member of IEEE TCDE executive committee, a steering committee member of DASFAA, WISE and ADC, and a steering committee chair of APWeb.

Position statement

TCDE is the flagship international society for data management and database systems researchers and practitioners. Should I have the honor to serve as the next TCDE Chair, I will continue the great work of the previous chairs, Paul Larson, David Lomet and Kyu-Young Whang, to work closely with the ICDE steering committee as well as the VLDB Endowment and ACM SIGMOD to ensure that ICDE maintains its status as a top quality international database conference, with the Data Engineering Bulletin editorial board to make it continuously to be a high value information source for all TCDE members. I will lead and work with the TCDE executive committee to promote TCDE membership and to build an energetic and healthy TCDE community.

Xiaofang Zhou
University of Queensland
Brisbane, Australia

Letter from the Special Issue Editor

This issue explores how machine learning (ML) has become an important application area for data management research and practice over the last few years.

The issue begins with two articles on the topic that has been at the forefront of the convergence of data management and ML research: probabilistic databases. When they wrote their revolutionary 2004 VLDB paper "Efficient Query Evaluation on Probabilistic Databases", I'm not sure whether or not Nilesh Dalvi and Dan Suciu realized how deeply connected their ideas were with the probabilistic structures that underlie much of modern ML, but those connections are becoming clear today. Fittingly, this issue opens with an article written by Eric Gribkoff, Dan Suciu, and Guy Van den Broeck that explains and explores some of these connections. They consider how the idea of lifted inference on probabilistic graphical models is nothing more than the evaluation of an SQL aggregate query in a probabilistic database, and how inference for ML can be performed using a probabilistic database based on this equivalence. The second article, by Dan Olteanu and Sebastiaan J. van Schaik, describes the ENFrame system. ENFrame is a so-called *probabilistic programming* platform that is fundamentally based upon ideas and algorithms that came out of research into probabilistic databases. "Probabilistic programming" is an exciting idea, still in its infancy, that is being pursued by researchers in several communities. The idea is that a user should be able to write code in a high-level programming language that manipulates uncertain or probabilistic data. The platform itself—and not the programmer—should be responsible for drawing inferences about the probability distributions induced.

The next article considers a promising ML-oriented use case for relational database-like engines: as platforms for very large-scale knowledge extraction. Chris Re et al. from Stanford describe DeepDive, which is a system for declaratively specifying (and running) large-scale knowledge-base construction workflows. While no one would describe DeepDive as a database system, it makes extensive use of ideas from database systems, including the use of SQL as a language to perform feature extraction, and the use of a relational engine to perform scalable statistical inference.

In the next article, Daisy Wang and her students at the University of Florida expand on this second idea, and consider how a database can be an excellent platform for running large-scale machine learning algorithms, with a particular emphasis on the learning of graphical models.

Recently, there has been a loud technical argument over the "correct" platform for implementing and running large-scale parallel or distributed machine learning codes, with many proposals from within and without the database community. The next two articles describe specific systems from within the database community for this task. The first, from IBM Almaden, describes SystemML, which is a programming language and platform for writing statistical codes, with an emphasis on supporting distributed linear algebra. The focus in the article here is on SystemML's optimizer, which takes as input a computation described in SystemML's programming language, and generates an efficient execution plan. The second article, from Brown, describes TupleWare, which is a database-like system with a focus on the efficient execution of UDFs. One of the central ideas in the TupeWare system is the compilation of high-level dataflow plans into low-level LLVM code, with the idea being that low-level code that can be executed directly is much more efficient than the classical, Volcano-style iterators.

The issue closes with an article from Duke that describes Cumulon. Deploying large-scale ML and statistical analysis in the cloud is difficult because of the bewildering array of choices facing an analyst: What machines to use? How much work should be given to each parallel task? How many tasks should be defined? Cumulon makes these choices using a database-like optimization process, where, in addition to figuring out the execution steps, the optimizer also figures out how to purchase and utilize the compute resources.

Taken together, these articles represent a broad sampling of some of the work at the intersection of databases, declarative systems, and ML. I hope you have as much fun reading the article as I did putting it together.

Chris Jermaine
Rice University

Lifted Probabilistic Inference: A Guide for the Database Researcher

Eric Gribkoff
University of Washington
eagribko@cs.uw.edu

Dan Suciu
University of Washington
suciu@cs.uw.edu

Guy Van den Broeck
UCLA
guyvdb@cs.ucla.edu

1 Introduction

Modern knowledge bases such as Yago [14], DeepDive [19], and Google's Knowledge Vault [6] are constructed from large corpora of text by using some form of supervised information extraction. The extracted data usually starts as a large probabilistic database, then its accuracy is improved by adding domain knowledge expressed as hard or soft constraints. Finally, the knowledge base can be queried using some general-purpose query language (SQL, or Sparql).

A key technical challenge during the construction, refinement, and querying of knowledge bases is probabilistic reasoning. Because of the size of the data involved, probabilistic reasoning in knowledge bases becomes a central data management problem. The number of random variables is very large, typically one for each fact in the knowledge base. Most systems today perform inference by using Markov Chain Monte Carlo (MCMC) methods; for example, DeepDive uses Gibbs Sampling, a form of MCMC. While MCMC methods are broadly applicable, probabilistic inference techniques based on MCMC have no polynomial time convergence guarantees; this is unavoidable, as exact probabilistic inference is provably intractable. Nevertheless, many classes of queries can be evaluated efficiently using an alternate, quite promising approach: *lifted inference*. While traditional methods first ground the knowledge base and run MCMC over the resulting large probabilistic space, lifted inference performs the inference directly on the first-order expression, which is much smaller than the entire probabilistic space. Lifted inference has evolved separately in the AI community [21] and in the database community [28]. When applicable, lifted inference is extremely efficient, in theory and in practice. Evaluating a query using lifted inference is equivalent to computing a SQL query with aggregate operators. Today's database engines have a large toolbox for computing aggregate queries, and such queries can be computed quite efficiently both on a single server and on a distributed system. But lifted inference has an important limitation: it only applies to some first-order expressions. In this paper we review lifted inference, both from the AI and from the database perspective, and describe some recent techniques that have expanded the applicability of lifted inference.

2 Probabilistic Databases

A Knowledge Base starts as a large collection of facts with probabilities, called a *probabilistic database* in the DB community. We review here the basic concepts following [28]. A probabilistic database is a relational

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

Tweeter

Name	Topic	P
Alice	Transactions	0.7
Alice	SocialNetworks	0.8
Carol	SocialNetworks	0.9

Follows

Follower	Followee	P
Alice	J.Bieber	0.3
Alice	G.Keillor	0.4
Carol	J.Bieber	0.5

Celebrity

Name	P
J.Bieber	0.99
M.Jordan	0.66
G.Keillor	0.33

Figure 1: A probabilistic database

database where each tuple is a random Boolean variable. Each relation has one extra attribute, P, representing the probability that the record is present. For now, we will assume that all tuples are independent, and will revisit this assumption in Section 5. The main research challenge in probabilistic databases is to compute the output probabilities for query answers.

A simple example is in Figure 1, which shows a tuple-independent probabilistic database with three tables. Consider the following SQL query:

```
select distinct Tweeter.topic
from Tweeter, Follows
where Tweeter.name = Follows.follower
```

Equivalently, the query written in First-Order Logic (FOL) is:

$$Q_1(t) = \exists x \exists y \, \text{Tweeter}(x, t) \wedge \text{Follows}(x, y)$$

The query retrieves all topics tweeted by people who are followers (occur in the `Follows` table). The answers to the query are uncertain, and the probabilistic database system needs to compute the output probability for each answer:

Topic	P
Transactions	$0.7 \cdot [1 - (1 - 0.3) \cdot (1 - 0.4)] = 0.406$
SocialNetworks	$1 - \{1 - 0.8 \cdot [1 - (1 - 0.3) \cdot (1 - 0.4)]\} \cdot \{1 - 0.9 \cdot 0.5\} = 0.7052$

In general, if a database has n tuples, then it defines a probability space with 2^n possible outcomes. While some queries, like the query above, can be computed efficiently, for other queries the complexity of computing the output probabilities is #P-hard in the size of the database. A simple example is the following query, whose complexity is #P-hard:

$$Q_2(t) = \exists x \exists y \, \text{Tweeter}(x, t) \wedge \text{Follows}(x, y) \wedge \text{Celebrity}(y)$$

One common misconception about probabilistic databases is that the independence assumption limits their usefulness. This is wrong: correlations can be modeled using constraints, and the latter folded back inside query evaluation over tuple-independent probabilistic databases; we will discuss this in Section 5.

3 Weighted Model Counting (WMC)

Probabilistic inference in Knowledge Bases can be reduced to the Weighted Model Counting (WMC) problem, which is a very well studied problem in the theory community. In Model Counting, or #SAT [11], one has to compute the number of models $\#F$ of a Boolean formula F. In WMC, each model has a weight and we have to compute the *sum of their weights*. Several state-of-the-art probabilistic inference algorithms for Bayesian networks [5, 24, 2], relational Bayesian networks [3] and probabilistic programs [8] perform a reduction to WMC. We give here a brief overview of WMC.

Propositional WMC Consider a Boolean Formula F over n Boolean variables $X_1, \ldots, X_n$, and two weight functions, $\bar{w}, w$, that associates to each Boolean variable X_i two weights: $\bar{w}(X_i), w(X_i) \in R$. The weight of a truth assignment, $\theta : X_1, \ldots, X_n \to 0, 1$, is defined as:

$$\text{WM}(\theta) = \prod_{i=1,n:\theta(X_i)=0} \bar{w}(X_i) \times \prod_{i=1,n:\theta(X_i)=1} w(X_i)$$

In other words, when $X_i = 0$ then we multiply with $\bar{w}(X_i)$, and when $X_i = 1$ then we multiply with $w(X_i)$. The *weight of the Boolean formula*, F, is the sum of weights of all assignments that satisfy F:

$$\text{WMC}(F, \bar{w}, w) = \sum_{\theta:X_1,\ldots,X_n \to 0,1,\theta(F)=1} \text{WM}(\theta)$$

For a simple illustration, if $F = X_1 \wedge X_2$ then $\text{WMC}(F, \bar{w}, w) = w(X_1) \cdot w(X_2) \cdot \prod_{i=3}^{n}(w(X_i) + \bar{w}(X_i))$.

Often, the weight $\bar{w}(X_i)$ is omitted from the specification of the WMC and assumed by default to be equal to 1. In Section 6 we will use values for $\bar{w}$ that are different from 1. There are three special cases of WMC of particular interest.

1. In the *Standard* WMC, $\bar{w}(X_i) = 1$ for all i. In that case we write $\text{WMC}(F, w)$ instead of $\text{WMC}(F, 1, w)$.
2. In *Model Counting*, #SAT, we set $w(X_i) = \bar{w}(X_i) = 1$ for all i, then $WMC(F, w) = \#F$, the number of satisfying assignments of F.
3. In *Probability Computation*, each variable X_i is set to true with some known probability $p(X_i) \in [0, 1]$, and we want to compute $\Pr(F, p)$, the probability that F is true. Then we set $\bar{w}(X_i) = 1 - p(X_i)$, $w(X_i) = p(X_i)$, in other words we have $\Pr(F, p) = \text{WMC}(F, p, 1 - p)$.

The WMC, the standard WMC, and the probability computation problems are equivalent, under the mild assumption that none of the following denominators are zero:

$$\begin{aligned} \text{WMC}(F, \bar{w}, w) &= \text{WMC}(F, 1, w/\bar{w}) \times \prod_i \bar{w}(X_i) \\ \text{WMC}(F, \bar{w}, w) &= \Pr(F, w/(w + \bar{w})) \times \prod_i (w(X_i) + \bar{w}(X_i)) \end{aligned} \tag{1}$$

Each of the three problems is reducible to the others. As such, we shall use them interchangeably. Probabilities are most convenient to describe lifted inference in Section 4, while weights are most convenient to describe the soft constraints in Section 5. Notice that weighted model count and probability computation strictly generalize #SAT.

In the original paper where Valiant introduced the class #P [29] he already noted that #SAT is #P-complete; he proved in a companion paper [30] that the problem remains #P-hard even if F is restricted to positive 2CNF formulas, and similarly for positive 2DNF formulas. Provan and Ball [22] proved that F can be further restricted to be a *Partitioned* Positive 2CNF (or 2DNF), meaning that every clause is of the form $X_i \vee Y_j$, where $X_1, X_2, \ldots$, and $Y_1, Y_2, \ldots$, are disjoint sets of variables, but it is possible that $X_i = X_j$ or $Y_i = Y_j$ for $i \neq j$. Of course, these classes of formulas remain #P-hard for WMC and for probabilistic computation.

First-Order WMC Next, we move from propositional formulas to sentences in first-order logic. Consider a relational vocabulary (database schema): $R_1, \ldots, R_k$. Assume a domain of constants D of size n, and denote $\mathcal{L}$ the set of all grounded atoms that can be built using the relations R_j and the values in the domain D. Let $\bar{w}, w : \mathcal{L} \to R^+$ be two weight functions. For any first-order sentence Φ, its *grounding*, or *lineage* is the Boolean

function $F_{\Phi,D}$ over the grounded atoms $\mathcal{L}$ (viewed as propositional Boolean variables), obtained by replacing quantifiers $\exists$ and $\forall$ with disjunctions or conjunctions over the domain D: $F_{\exists x\Phi,D} = \bigvee_{a\in D} F_{\Phi[a/x],D}$, and $F_{\forall x\Phi,D} = \bigwedge_{a\in D} F_{\Phi[a/x],D}$; furthermore, $F_{\Phi_1 \text{ op } \Phi_2,D} = F_{\Phi_1,D}$ op $F_{\Phi_2,D}$ for op $\in \vee, \wedge$, $F_{\neg\Phi,D} = \neg F_{\Phi,D}$ and finally $F_t = t$ for a ground atom $t \in \mathcal{L}$. The grounding of even a simple FO sentence can be quite large. Consider the sentence $\Phi = \exists x, y, z, w,\ R_1(x, y) \wedge R_2(y, z) \wedge R_3(z, w)$. If the domain D has size 1000, the grounding of Φ contains 10^{12} distinct formulas. The terms "grounding" and "lineage" were introduced independently in the AI literature and in the DB literature; we will use them interchangeably in this paper.

The weighted *first-order* model count of sentence Φ for domain D is the weighted model count of its grounding, that is, $\text{WFOMC}(\Phi, D, \bar{w}, w) = \text{WMC}(F_{\Phi,D}, \bar{w}, w)$.

Definition 1 (The WFOMC Problem:): Given a domain D, weight functions $w, \bar{w}$, and first-order sentence Φ, compute $\text{WFOMC}(\Phi, D, \bar{w}, w)$.

One can view the triple $(D, w, \bar{w})$ as a probabilistic database, where every tuple t has probability $w(t)/(w(t) + \bar{w}(t))$, and the FO-WMC is equivalent to the query computation problem in a probabilistic database.

4 Lifted Inference

The term *lifted inference* was coined by Poole [21] to mean (informally) inference techniques that can solve WFOMC without grounding the FO sentence. Subsequently different classes of techniques have been called "lifted", even with grounding. A more rigorous definition was proposed by Van den Broeck [31]; an algorithm is called *domain lifted* if it runs in time polynomial in n, the size of the domain. Independently, the database community searched for algorithms whose data complexity is in PTIME. Following Vardi [36], in *data complexity* the query expression is fixed, and one measures the complexity as a function of the size of the input database. Dalvi and Suciu [4] showed that some queries have a PTIME data complexity, while others are #P-hard. For the WFOMC task, these two notions developed in the AI and DB communities coincide:[1] liftability, defined as an algorithm that runs in PTIME in the size of the domain, and PTIME data complexity. We briefly review here lifted inference algorithms for WFOMC.

Lifted inference algorithms compute $\Pr(\Phi)$ by repeatedly applying one of the following rules to a FO sentence Φ, until it becomes a ground tuple t, in which case $\Pr(t)$ can be simply looked up in the database:

Independent (decomposable) AND $\Pr(\Phi_1 \wedge \Phi_2) = \Pr(\Phi_1) \cdot \Pr(\Phi_2)$, when the lineages of Φ_1, Φ_2 have disjoint sets of atoms. This pre-condition is checked by analysis of the first-order expressions and implies Φ_1, Φ_2 are independent.

Independent (decomposable) FORALL $\Pr(\forall x\Phi) = \prod_{a\in D} \Pr(\Phi[a/x])$ if x is a "separator variable" in Φ (for any two constants $a \neq b$, $\Phi[a/x]$ is independent from $\Phi[b/x]$).

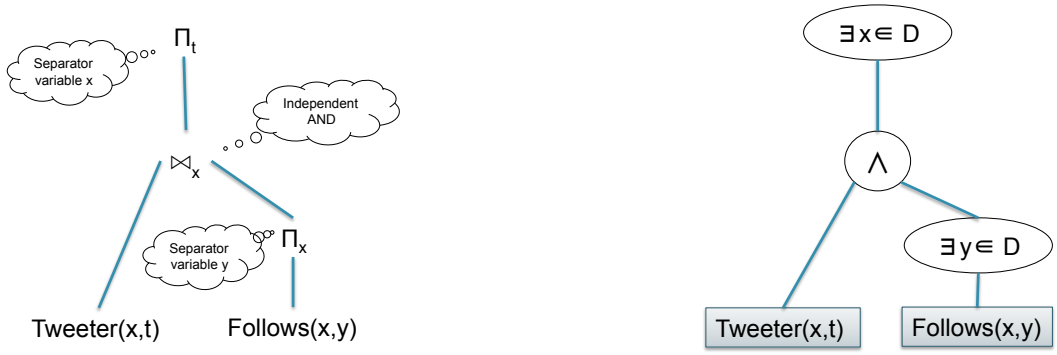
Inclusion/exclusion $\Pr(\Phi_1 \vee \Phi_2) = \Pr(\Phi_1) + \Pr(\Phi_2) - \Pr(\Phi_1 \wedge \Phi_2)$.

Negation $\Pr(\neg\Phi) = 1 - \Pr(\Phi)$

Exclusive (deterministic) OR $\Pr(\Phi_1 \vee \Phi_2) = \Pr(\Phi_1) + \Pr(\Phi_2)$ if Φ_1, Φ_2 are exclusive ($\Phi_1 \wedge \Phi_2 \equiv$ false).

By duality we also have $\Pr(\Phi_1 \vee \Phi_2) = 1 - (1 - \Pr(\Phi_1) \cdot (1 - \Pr(\Phi_2)))$ when Φ_1, Φ_2 are independent, and $\Pr(\exists x\Phi) = 1 - \prod_{a\in D}(1 - \Pr(\Phi[a/x]))$ when x is a separator variable in Φ. The rules are effective, in the sense that one can check their preconditions using a simple syntactic analysis on the FOL expressions, and applying the rules can be done in time polynomial in the size of the domain D [28]. For a simple example, to compute

[1] In other contexts (e.g., Markov logic), one may want to distinguish between the data and domain, resulting in distinct notions.



(a) (b)

Figure 2: A Query Plan (a) and the equivalent FO-d-DNNF circuit (b); both compute the query Q_1 in Section 2. The Appendix shows how to express the query directly in Postgres.

the probability of $\exists x \exists y \text{Tweeter}(x, \text{SocialNetworks}) \wedge \text{Follows}(x, y)$ on the database in Figure 1 one applies the separator variable rules to $\exists x$, followed by an independent AND, arriving at the formula in Section 2.

A sequence of lifted inference rules is naturally represented as a tree, which can be interpreted either as a *query plan*, or as a *circuit*. Figure 2 shows the Relational Query plan [28], and the equivalent FO-d-DNNF circuit [35, 32] for computing the query Q_1 in Section 2.

For some queries one cannot compute their probabilities using lifted inference alone; for example this must happen when the query has a data complexity that is #P-hard. For example, consider

$$\Phi_2 = \exists x \exists y \text{Tweeter}(x, \text{SocialNetworks}) \wedge \text{Follows}(x, y) \wedge \text{Celebrity}(y) \tag{2}$$

which corresponds to Q_2 in Section 2. Its lineage is the following (where we abbreviate the constants in the domain by their first letter, e.g. writing A instead of Alice, etc):

$$\begin{aligned} F = &\text{Tweeter}(A, S) \wedge \text{Follows}(A, J) \wedge \text{Celebrity}(J) \\ &\vee \text{Tweeter}(A, S) \wedge \text{Follows}(A, G) \wedge \text{Celebrity}(G) \\ &\vee \text{Tweeter}(C, S) \wedge \text{Follows}(A, J) \wedge \text{Celebrity}(J) \end{aligned}$$

It is not hard to see that computing $\Pr(\Phi_2)$ is #P-hard in the size of D: indeed, any PP2DNF formula $\bigvee X_i \wedge Y_j$ is equivalent to a lineage above, if we choose Tweeter to represent the variables X_i, Celebrity to represent Y_j, and Follows to represent all prime implicants $X_i \wedge Y_j$ (by setting the probabilities to 1 or 0). In other words, computing the answer to Q_2 is #P-hard, and therefore (under standard complexity assumptions) its probability cannot be computed using lifted inference.

Completeness One open question is whether the lifted inference rules are complete for exact probabilistic inference: one wonders if one should search for new rules, or whether our list is exhaustive in some formal sense. The criteria for "completeness" is to be able to compute the probability of any query computable in PTIME. It has been shown in [4] that the rules are complete to evaluate all Unions of Conjunctive Queries

(UCQ) that are in PTIME. It follows immediately that the rules are complete both for UCQ, and for their dual counterpart, the positive, universal fragment of First Order Logic. However, these rules are *not* complete for languages with negation: as the following example shows, there are PTIME queries with negation where none of the above rules apply, and an algorithm incorporating only these rules is immediately stuck. Applying logical resolution can derive equivalent queries such that the above rules apply, but even then it is open if this gives a complete algorithm for lifted inference [13]. For example, consider this query:

$$\forall x \forall y (\text{Tweeter}(x) \vee \neg \text{Follows}(x, y)) \wedge (\text{Follows}(x, y) \vee \neg \text{Celebrity}(y))$$

(Written as $\text{Follows}(x, y) \Rightarrow \text{Tweeter}(x)$ and $\text{Celebrity}(y) \Rightarrow \text{Follows}(x, y)$, it says that every follower is a tweeter, and that every celebrity is followed by everyone.) Our list of rules are stuck on this query: in fact, if we remove all negations, then the query is known to be #P-hard, hence to get unstuck we need to use the negation in an essential way. We do this by applying resolution between the two clauses then we obtain the equivalent query:

$$\begin{aligned}
\Phi =& \forall x \forall y (\text{Tweeter}(x) \vee \neg \text{Follows}(x, y)) \wedge (\text{Follows}(x, y) \vee \neg \text{Celebrity}(y)) \wedge (\text{Tweeter}(x) \vee \neg \text{Celebrity}(y)) \\
\equiv& \forall x \forall y (\text{Tweeter}(x) \vee \neg \text{Follows}(x, y)) \wedge (\text{Follows}(x, y) \vee \neg \text{Celebrity}(y)) \wedge \text{Tweeter}(x) \\
\vee& \forall x \forall y (\text{Tweeter}(x) \vee \neg \text{Follows}(x, y)) \wedge (\text{Follows}(x, y) \vee \neg \text{Celebrity}(y)) \wedge \neg \text{Celebrity}(y) \\
\equiv& \forall x \forall y \wedge (\text{Follows}(x, y) \vee \neg \text{Celebrity}(y)) \wedge \text{Tweeter}(x) \\
\vee& \forall x \forall y (\text{Tweeter}(x) \vee \neg \text{Follows}(x, y)) \wedge \neg \text{Celebrity}(y)
\end{aligned}$$

Now we apply inclusion/exclusion and obtain:

$$\begin{aligned}
\Pr(\Phi) = & \Pr(\forall x \forall y (\text{Follows}(x, y) \vee \neg \text{Celebrity}(y)) \wedge \text{Tweeter}(x)) \\
& + \Pr(\forall x \forall y (\text{Tweeter}(x) \vee \neg \text{Follows}(x, y)) \wedge \neg \text{Celebrity}(y)) \\
& - \Pr(\forall x \text{Tweeter}(x) \wedge \forall y \neg \text{Celebrity}(y))
\end{aligned}$$

and all three resulting expressions can be computed using lifted inference.

Symmetric Probabilities A special case is when all ground tuples belonging to the same relation have the same probabilities. These types of weights occur in the #SAT model counting setting, and are of particular interest in machine learning. Van den Broeck [34] has established recently that, if one adds one other rule (to eliminate unary predicates) then the rules are complete when all probabilities are symmetric, and the sentence Φ has at most two logical variables. In that case, for example, query (2) is computable in PTIME.

Rule independence Are all rules necessary? In other words, can we remove some rules form the list because we can express them using the other rules? For example, exclusive OR can be expressed using inclusion/exclusion. An interesting open question is whether inclusion/exclusion can be substituted by exclusive OR.

Approximate Lifted Inference Lifted inference has been extended beyond exact inference to handle approximations. In the database community, Dylla [7] reasons on the FOL formula to determine lower and upper bounds on the output probability, extending the techniques in [20] developed for propositional inference. The AI community has a rich literature on approximate lifted inference with symmetric probabilities (e.g., [18, 16, 27, 33, 10, 26]).

Other Lifted Tasks Next to the task of computing marginal probabilities, the AI community considers several other inference and learning tasks. This includes the *Maximum a Posteriori* (MAP) problem, that is, to determine the most likely state of all variables in the distribution. For recent work on lifting the MAP task with symmetric probabilities, see [17, 1, 25]. Because the standard probabilistic database setting assumes tuple independence, the MAP task has received little attention in the probabilistic database literature: without any additional constraints, the probability distribution completely factorizes, and computing the MAP state reduces to setting every tuple with probability greater than 0.5 to true and the rest to false. Recently, [12] proposed the *Most Probable Database* (MPD) problem, which is variation on the MAP task that has several database applications. Nevertheless, exact lifted inference for the MAP and MPD tasks is still poorly understood. Other tasks one may consider include lifted marginal MAP, a harder generalization of the MAP task where the most likely state of only some of the variables is sought, and lifted structure and parameter learning. These are largely unexplored problem areas.

5 Hard and Soft Constraints

Conceptually, the construction of a large Knowledge Base has two major steps. The first step generates a database of facts, by using some information extraction systems. The result is a database where each fact has some confidence, or probability, and, thus, it looks much like a probabilistic database. Second, one adds constraints representing common knowledge; these constraints, also called rules, are FOL formulas and may be either hard, or soft.

We illustrate soft rules by adapting the syntax of Markov Logic Networks (MLNs), introduced by Richardson and Domingos [20]. Consider the following three soft rules:

$$
\begin{aligned}
5.38: &\quad \Phi_1(x) = \text{Tweeter}(x, \text{MachineLearning}) \Rightarrow \text{Follows}(x, \text{M.Jordan}) \\
1.2: &\quad \Phi_2(x) = \forall y(\text{Follows}(x, y) \Rightarrow \text{Celebrity}(y)) \\
0.8: &\quad \Phi_3(x) = \forall y \forall z(\text{Tweeter}(x, y) \wedge \text{Tweeter}(x, z) \wedge y \neq z)
\end{aligned}
$$

The first rule says that tweeters on MachineLearning are likely to follow M.Jordan. The second rule says that people who are followed are typically celebrities. Note that both these rules have a weight > 1, and therefore we assert that they are more likely to happen. The last rule has a weight < 1, saying that for any tweeter x, it is unlikely that x tweets on two different topics $y \neq z$. Each formula has a free variable x.

An MLN is a set of FOL formulas $\text{MLN} = \Phi_1, \ldots, \Phi_m$ and two weight functions $\bar{w}(\Phi_i), w(\Phi_i)$; the weight function $\bar{w}(\Phi_i)$ is often equal to 1, and omitted from the specification of the MLN. The formulas may have free variables, in other words they are not necessarily sentences; we write $\Phi_i(\mathbf{x}_i)$ to denote that $\mathbf{x}_i$ are the free variables in Φ_i.

Given a domain D, we will describe now how the MLN defines a probability space on the subsets $\mathcal{L}$. Recall that $\mathcal{L}$ represents the set of grounded atoms over the domain D. A subset of $\mathcal{L}$ is called a *possible world*, and we will define it equivalently as a truth assignment $\theta : \mathcal{L} \to 0, 1$. Its *weight* is:

$$
\text{WM}(\theta) = \prod_{\Phi_i \in \text{MLN}, \mathbf{a} \in D^{|\mathbf{x}_i|} : \theta \not\models \Phi_i[\mathbf{a}/\mathbf{x}_i]} \bar{w}(\Phi_i) \times \prod_{\Phi_i \in \text{MLN}, \mathbf{a} \in D^{|\mathbf{x}_i|} : \theta \models \Phi_i[\mathbf{a}/\mathbf{x}_i]} w(\Phi_i)
$$

In other words, to compute the weight of a possible world θ, we take every formula $\Phi_i(\mathbf{x}_i)$ in the MLN, substitute $\mathbf{x}_i$ in all possible ways with constants $\mathbf{a}$ from the domain to obtain a sentence $\Phi_i[\mathbf{a}/\mathbf{x}_i]$, and we multiply the weight either by $\bar{w}(\Phi_i)$, or by $w(\Phi_i)$, depending on whether the sentence $\Phi_i[\mathbf{a}/\mathbf{x}_i]$ is false or true in the world θ. For example, consider a simple MLN with the single rule $\Phi(x) = \exists y R(x, y)$, weight functions $w(\Phi) = 2$ and $\bar{w}(\Phi) = 0.5$, with domain $x, y \in \{A, B, C\}$. Let θ be a world where $R(A, A), R(A, B), R(B, A)$ are true

and $R(A,C), R(B,B), R(B,C), R(C,A), R(C,B), R(C,C)$ are false. Then $\Phi(A), \Phi(B)$ are true and $\Phi(C)$ is false, hence $\text{WM}(\theta) = 2 \cdot 2 \cdot 0.5 = 2$.

The weight of a query (sentence) Φ is defined as before:

$$\text{WMC}(\Phi, \text{MLN}, \bar{w}, w) = \sum_{\theta:\mathcal{L}\to 0,1:\theta(F_\Phi)=1} \text{WM}(\theta) \tag{3}$$

and its probability is obtained by dividing by a normalization factor Z:

$$\Pr(\Phi) = \frac{\text{WMC}(\Phi, \text{MLN}, \bar{w}, w)}{Z}$$
$$\text{where } Z = \sum_{\theta:\mathcal{L}\to 0,1} \text{WM}(\theta)$$

In a typical Knowledge Base one starts with a tuple-independent probabilistic database, then adds soft rules that represent common knowledge. Once we add rules, the tuples in the database are no longer independent. In fact, every grounding for some formula in the MLN creates a new factor, and correlates all the tuples that occur in that grounding.

In the standard definition of an MLN, every formula has $\bar{w}(\Phi_i) = 1$, in other words we have only the positive weight function $w(\Phi_i)$. A "non-standard" MLN, which has explicit negated weights $\bar{w}(\Phi_i)$, can be easily converted into a standard MLN by using equation (1). The negated weights $\bar{w}$, however, make it easier to represent hard constraints by setting $\bar{w}(\Phi_i) = 0$ and $w(\Phi_i) = 1$: in this case the formula Φ_i acts as a hard constraint, because only worlds where Φ_i is true contribute to the sum (3). In the standard formulation, a hard constraint is represented by setting $w(\Phi_i) = \infty$, requiring the system to handle hard constraints separately.

6 Recent Developments: Transformations and Negative Weights

In a series of recent papers [15, 34] some researchers (including the authors) have found simple transformations that convert between MLNs and WFOMC problems, and use clever expressions on the weights in order to simplify the logical formulas. The new weights may be negative, which may correspond to either negative probabilities, or probabilities > 1. (Recall that the connection between weights w, $\bar{w}$ and the probability p is $p = w/(w + \bar{w})$.) We give here a brief overview of these rewritings.

First we show a well known fact, that every MLN can be converted into an equivalent tuple-independent probabilistic database, plus hard constraints [35, 9]. If $\text{MLN} = \Phi_1(\mathbf{x}_1), \ldots, \Phi_m(\mathbf{x}_m)$, then for each $i = 1, m$ we create a new relational symbol T_i of arity $|\mathbf{x}_i|$, add the hard constraint:

$$\Gamma_i = \forall \mathbf{x}_i (\Phi_i(\mathbf{x}_i) \Leftrightarrow T(\mathbf{x}_i)) \tag{4}$$

and create a new probabilistic relation T_i where all tuples $T_i(\mathbf{a})$ have the same weights:

$$\bar{w}(T_i) = \bar{w}(\Phi_i) \qquad\qquad w(T_i) = w(\Phi_i)$$

(We write $w(T_i)$ instead of $w(T_i(\mathbf{a}))$ since all tuples in T_i have exactly the same weights.) In other words, we move the weights from the formula Φ_i to the new relation T_i, and add a hard constraint asserting that the formula and the relation are equivalent.

A disadvantage of Eq.(4) is that it introduces negations, because it expands to $(\neg\Phi_i \vee T_i) \wedge (\Phi_i \vee \neg T_i)$. As we explained earlier, negations are more difficult to handle in lifted inference. A way around this is to modify the hard constraint to:

$$\Gamma_i = \forall \mathbf{x}_i (\Phi_i(\mathbf{x}_i) \vee T_i(\mathbf{x}_i))$$

and set the weights of each tuple $T_i(\mathbf{a})$ with the objective of achieving the equalities:

$$\bar{w}(\Phi_i) = w(T_i) \qquad\qquad w(\Phi_i) = \bar{w}(T_i) + w(T_i)$$

Recall that $\bar{w}(\Phi_i)$ is often 1, and in that case $w(T_i) = 1$. These expressions are justified by the following argument. Consider some values $\mathbf{a}$ for the variables $\mathbf{x}_i$. If $\Phi_i[\mathbf{a}/\mathbf{x}_i]$ is false in a world, then $T_i[\mathbf{a}/\mathbf{x}_i]$ must be true (because $\Phi_i[\mathbf{a}/\mathbf{x}_i] \vee T_i[\mathbf{a}/\mathbf{x}_i]$ is a hard constraint), and the same factor $\bar{w}(\Phi_i) = w(T_i)$ is applied to this world in the original MLN as in the new MLN. If $\Phi_i[\mathbf{a}/\mathbf{x}_i]$ is true in a world, then $T_i[\mathbf{a}/\mathbf{x}_i]$ can be either true or false, and the factors of the two possible worlds add up to $\bar{w}(T_i) + w(T_i) = w(\Phi_i)$. Next, we use these two expressions to solve for $\bar{w}(T_i)$ and $w(T_i)$:

$$\bar{w}(T_i) = w(\Phi_i) - \bar{w}(\Phi_i) \qquad\qquad w(T_i) = \bar{w}(\Phi_i)$$

In other words, we have replaced the constraint (4) with a simpler constraint, without negations, by using more clever expressions for the weights. Notice that $\bar{w}(T_i)$ may be negative.

In fact, negation can be completely removed from the formula during lifted inference, through the following simple rewriting. For every relational symbol R that occurs negated in a formula Φ, create two new relational symbols of the same arities, N, T, replace all atoms of the form $\neg R(\mathbf{x})$ with $N(\mathbf{x})$ and add the following hard constraints:

$$\Gamma = \forall \mathbf{x}(R(\mathbf{x}) \vee N(\mathbf{x})) \wedge (R(\mathbf{x}) \vee T(\mathbf{x})) \wedge (N(\mathbf{x}) \vee T(\mathbf{x}))$$

with the following weights:

$$\bar{w}(N) = w(N) = w(T) = 1 \qquad\qquad \bar{w}(T) = -1$$

To see why this works, call a possible world "good" if, for every tuple $\mathbf{a}$, exactly one of the tuples $R(\mathbf{a})$ and $N(\mathbf{a})$ is true. In other words, in a good world N denotes $\neg R$. The hard constraint Γ further ensures that all tuples $T(\mathbf{a})$ are true in a good world, hence good worlds have the same weights in the original and in the new MLN. Consider now a "bad" world that satisfies Γ: then there exists a tuple $\mathbf{a}$ such that both $R(\mathbf{a})$ and $N(\mathbf{a})$ are true in that world. However, in that case $T(\mathbf{a})$ can be set to either true or false, and the weights of the two corresponding worlds cancel each other out. Therefore, the weights of all bad worlds cancel each other out.

Finally, another source of complexity in lifted inference comes from formulas that use a mixture of existential and universal variables. These can also be removed by using clever expressions for the weights. For example, consider a hard constraint:

$$\Phi = \forall x \exists y \varphi(x, y)$$

Create a new relational symbol $T(x)$, and replace Φ with the following constraint:

$$\Phi' = \forall x \forall y (T(x) \vee \neg \varphi(x, y))$$

and weights:

$$\bar{w}(T) = -1 \qquad\qquad w(T) = 1$$

We invite the reader to verify the correctness of this rewriting.

7 Open Problems

Many inference problems in AI are symmetric, where all tuples in a single relation have identical probabilities. As discussed in Section 4, the complexity of inference with symmetric probabilities is different (strictly easier) than in the asymmetric setting. In particular, many of the intractability results for probabilistic databases do not apply directly to the symmetric setting. It is of theoretical and practical interest to further characterize the complexity of queries with symmetric probabilities.

An open question is whether the inclusion/exclusion rule is necessary for lifted inference, or if some sequence of applications of the deterministic OR and other simpler rules is sufficient to compute any liftable query. Lifted inference rules for probabilistic databases use inclusion/exclusion, but in the AI community, lifted inference relies on deterministic OR. It is conjectured that inclusion/exclusion is strictly more powerful than deterministic OR: that is, there exist queries which cannot be computed in PTIME unless inclusion/exclusion is applied to identify intractable subqueries that cancel out and hence need not be computed. If the conjecture is correct, approaches that rely solely on deterministic OR can not be complete for lifted inference. However, deterministic OR is simpler to express; if it is equally as powerful, it may be preferred in practical systems.

As we have presented, there have been many recent advances in lifted inference techniques. There is a need for these new techniques to be incorporated into a robust, state of the art WFOMC tool for lifted inference that operates equally well in the database and AI settings. Effort in this direction is ongoing. For a prototype that implements many of the techniques discussed above, see `http://dtai.cs.kuleuven.be/wfomc`.

References

[1] H. Bui, T. Huynh, and S. Riedel. Automorphism groups of graphical models and lifted variational inference. In *Proceedings of UAI*, 2013.

[2] M. Chavira and A. Darwiche. On probabilistic inference by weighted model counting. *Artificial Intelligence*, 172(6-7):772–799, Apr. 2008.

[3] M. Chavira, A. Darwiche, and M. Jaeger. Compiling relational Bayesian networks for exact inference. *International Journal of Approximate Reasoning*, 42(1-2):4–20, May 2006.

[4] N. Dalvi and D. Suciu. The dichotomy of probabilistic inference for unions of conjunctive queries. *Journal of the ACM (JACM)*, 59(6):30, 2012.

[5] A. Darwiche. A logical approach to factoring belief networks. *Proceedings of KR*, pages 409–420, 2002.

[6] X. Dong, E. Gabrilovich, G. Heitz, W. Horn, N. Lao, K. Murphy, T. Strohmann, S. Sun, and W. Zhang. Knowledge vault: A web-scale approach to probabilistic knowledge fusion. In *KDD*, 2014.

[7] M. Dylla, I. Miliaraki, and M. Theobald. Top-k query processing in probabilistic databases with non-materialized views. In *ICDE*, pages 122–133, 2013.

[8] D. Fierens, G. Van den Broeck, I. Thon, B. Gutmann, and L. De Raedt. Inference in probabilistic logic programs using weighted CNF's. In *Proceedings of UAI*, pages 211–220, July 2011.

[9] V. Gogate and P. Domingos. Probabilistic theorem proving. In *UAI*, pages 256–265, 2011.

[10] V. Gogate, A. K. Jha, and D. Venugopal. Advances in lifted importance sampling. In *AAAI*, 2012.

[11] C. P. Gomes, A. Sabharwal, and B. Selman. Model counting. *Handbook of Satisfiability*, 185:633–654, 2009.

[12] E. Gribkoff, G. Van den Broeck, and D. Suciu. The most probable database problem. *Big Uncertain Data Workshop*, 2014.

[13] E. Gribkoff, G. Van den Broeck, and D. Suciu. Understanding the complexity of lifted inference and asymmetric weighted model counting. In *UAI*, pages 280–289, 2014.

[14] J. Hoffart, F. M. Suchanek, K. Berberich, and G. Weikum. Yago2: A spatially and temporally enhanced knowledge base from wikipedia. *Artif. Intell.*, 194:28–61, 2013.

[15] A. K. Jha and D. Suciu. Probabilistic databases with markoviews. *PVLDB*, 5(11):1160–1171, 2012.

[16] K. Kersting, B. Ahmadi, and S. Natarajan. Counting belief propagation. In *Proceedings of the Twenty-Fifth Conference on Uncertainty in Artificial Intelligence*, pages 277–284. AUAI Press, 2009.

[17] M. Mladenov, B. Ahmadi, and K. Kersting. Lifted linear programming. In *International Conference on Artificial Intelligence and Statistics*, pages 788–797, 2012.

[18] M. Niepert. Symmetry-aware marginal density estimation. In *AAAI*, 2013.

[19] F. Niu, C. Zhang, C. Ré, and J. W. Shavlik. Deepdive: Web-scale knowledge-base construction using statistical learning and inference. In *VLDS*, pages 25–28, 2012.

[20] D. Olteanu and H. Wen. Ranking query answers in probabilistic databases: Complexity and efficient algorithms. In *ICDE*, pages 282–293, 2012.

[21] D. Poole. First-order probabilistic inference. In *Proceedings of IJCAI*, pages 985–991, 2003.

[22] J. S. Provan and M. O. Ball. The complexity of counting cuts and of computing the probability that a graph is connected. *SIAM Journal on Computing*, 12(4):777–788, 1983.

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

[24] T. Sang, P. Beame, and H. Kautz. Solving Bayesian networks by weighted model counting. In *Proceedings of AAAI*, volume 1, pages 475–482, 2005.

[25] S. Sarkhel, D. Venugopal, P. Singla, and V. Gogate. Lifted map inference for markov logic networks. In *AISTATS*, pages 859–867, 2014.

[26] P. Sen, A. Deshpande, and L. Getoor. Bisimulation-based approximate lifted inference. In *Proceedings of the Twenty-Fifth Conference on Uncertainty in Artificial Intelligence*, pages 496–505. AUAI Press, 2009.

[27] P. Singla and P. Domingos. Lifted first-order belief propagation. In *AAAI*, volume 8, pages 1094–1099, 2008.

[28] D. Suciu, D. Olteanu, C. Ré, and C. Koch. Probabilistic databases. *Synthesis Lectures on Data Management*, 3(2):1–180, 2011.

[29] L. G. Valiant. The complexity of computing the permanent. *Theor. Comput. Sci.*, 8:189–201, 1979.

[30] L. G. Valiant. The complexity of enumeration and reliability problems. *SIAM Journal on Computing*, 8(3):410–421, 1979.

[31] G. Van den Broeck. On the completeness of first-order knowledge compilation for lifted probabilistic inference. In *NIPS*, pages 1386–1394, 2011.

[32] G. Van den Broeck. *Lifted Inference and Learning in Statistical Relational Models*. PhD thesis, KU Leuven, 2013.

[33] G. Van den Broeck, A. Choi, and A. Darwiche. Lifted relax, compensate and then recover: From approximate to exact lifted probabilistic inference. In *Proceedings of UAI*, 2012.

[34] G. Van den Broeck, W. Meert, and A. Darwiche. Skolemization for weighted first-order model counting. In *Proceedings of KR*, 2014.

[35] G. Van den Broeck, N. Taghipour, W. Meert, J. Davis, and L. De Raedt. Lifted probabilistic inference by first-order knowledge compilation. In *Proceedings of IJCAI*, pages 2178–2185, 2011.

[36] M. Y. Vardi. The complexity of relational query languages (extended abstract). In *STOC*, pages 137–146, 1982.

A Appendix

Below is the SQL code for running query Q_1 on the probabilistic database in Figure 1. The SQL query below implements the plan in Figure 2 and has been tested on PostgreSQL 9.2.3.

```
-- define an aggregate function to compute the product
create or replace function combine_prod (float, float) returns float as 'select $1 * $2' language SQL;
create or replace function final_prod (float) returns float as 'select $1' language SQL;
drop aggregate if exists prod (float);
create aggregate prod (float)
(  sfunc = combine_prod,
   stype = float,
   finalfunc = final_prod,
   initcond = '1.0'
);

-----------------------------------------------------
-- create/populate tables
create table Tweeter(name text, topic text, p float);
create table Follows(follower text, followee text, p float);

insert into Tweeter values('Alice', 'Transactions',   0.7);
insert into Tweeter values('Alice', 'SocialNetworks', 0.8);
insert into Tweeter values('Carol', 'SocialNetworks', 0.9);

insert into Follows values('Alice', 'J.Bieber', 0.3);
insert into Follows values('Alice', 'G.Keillor', 0.4);
insert into Follows values('Carol', 'J.Bieber', 0.5);


-----------------------------------------------------
-- run the query
with Temp as
  (select Follows.follower, 1.0-prod(1.0-p) as p
   from Follows
   group by Follows.follower)
select Tweeter.topic, 1.0-prod(1-Tweeter.p*Temp.p)
from Tweeter, Temp
where Tweeter.name=Temp.follower
group by Tweeter.topic;
```

Probabilistic Data Programming with ENFrame

Dan Olteanu and Sebastiaan J. van Schaik
University of Oxford

Abstract

This paper overviews ENFrame, a programming framework for probabilistic data. In addition to relational query processing supported via an existing probabilistic database management system, ENFrame allows programming with loops, assignments, conditionals, list comprehension, and aggregates to encode complex tasks such as clustering and classification of probabilistic data.

We explain the design choices behind ENFrame, some distilled from the wealth of work on probabilistic databases and some new. We also highlight a few challenges lying ahead.

1 Motivation and Scope

Probabilistic data management has gone a long, fruitful way in the last decade [20]: We have a good understanding of the space of possible relational and hierarchical data models and its implication on query tractability; the community already delivered several open-source systems that exploit the first-order structure of database queries for scalable inference, e.g., MystiQ [3], Trio [22], MayBMS/SPROUT [11], and PrDB [19] to name very few, and applications in the space of web data management [8, 6]. Significantly less effort has been spent on supporting complex data processing beyond mere querying, such as general-purpose programming.

There is a growing need for computing frameworks that allow users to build applications feeding on uncertain data without worrying about the underlying uncertain nature of such data or the computationally hard inference task that comes along with it. For tasks that only need to query probabilistic data, existing probabilistic database systems do offer a viable solution [20]. For more complex tasks, however, successful development requires a high level of expertise in probabilistic databases and this hinders the adoption of existing technology as well as communication between potential users and experts.

A similar observation has been recently made in the areas of machine learning [5] and programming languages [10]. Developing programming languages that allow probabilistic models to be expressed concisely has become a hot research topic [18]. Such programming languages can be imperative (C-style) or declarative (first-order logic), with the novelty that they allow to express probability distributions via generative stochastic models, to draw values at random from such distributions, and to condition values of program variables on observations. For inference, the programs are usually grounded to Bayesian networks and fed to MCMC methods [15]. In the area of databases, MCDB [13] and SimSQL [4] have been visionary in enabling stochastic analytics in the database by coupling Monte Carlo simulations with declarative SQL extensions and parallel database techniques.

The thesis of this work is that one can build powerful and useful probabilistic data programming frameworks that *leverage* existing work on probabilistic databases. ENFrame [21] is a framework that aims to fit this vision:

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

- Its programming language is a fragment of Python with constructs such as bounded-range loops, if-then-else statements, list comprehension, aggregates, variable assignments, and query calls to external database engines. A user program can express complex tasks such as clustering and classification intermixed with structured querying.

- The users are oblivious to the probabilistic nature of the input data: They program as if the input data were plain relational, with no uncertainty or layout intricacy. It is the job of ENFrame to make sense of the underlying data formats, probabilities, and input correlations, thus allowing low-level entry for users without expert knowledge on inference and probabilistic models.

  ENFrame relies on the Πgora probabilistic data integration system [17] for exposing a uniform relational view of the underlying data, which may be Bayesian networks for some sources and probabilistic c-tables for others (or special cases such as tuple-independent or block-independent disjoint tables).

- ENFrame adheres to the possible worlds semantics for its whole processing pipeline. Under this semantics, the input is a probability distribution over a finite set of possible worlds, with each world defining a database. The result of a user program is equivalent to executing it within each world and is thus a probability distribution over possible outcomes of the program variables. This distribution can be inspected by the user and can serve as input to probabilistic database queries and subsequent ENFrame programs.

- ENFrame uses a rich language of probabilistic events to symbolically express input correlations, trace the correlations introduced during computation, and enable result explanation, sensitivity analysis [14], incremental maintenance, and knowledge compilation-based approaches for approximate inference [9].

  While propositional formulas over random variables are expressive enough to capture computation traces of ENFrame programs, they can be very large and expensive to manage. ENFrame's event language extends algebraic formalisms based on semimodules for probabilistic data [2, 7] that can succinctly[1] encode probabilistic events with constructs mirroring those in the user language.

- ENFrame exploits the structure of queries and programs for efficient inference; it relies on SPROUT for query processing on probabilistic data [16, 9].

  To avoid iterating over all possible worlds, ENFrame employs approximation schemes and exploits the fact that many of the possible worlds are alike. To further speed up inference on networks of highly interconnected probabilistic events, such as those for clustering and classification programs, ENFrame uses parallel algorithms that exploit multi-core architectures.

There are key differences that set ENFrame apart from the myriad of recent probabilistic programming [18] and probabilistic data processing [1] approaches:

- ENFrame uses the semantics and a probabilistic data model compatible with probabilistic databases [20]. Its input can consist of arbitrarily correlated (and not only independent) probabilistic events. This enables processing pipelines mixing programming (via ENFrame) and querying (via SPROUT).

- The user need not be aware of the probabilistic nature of data or the possibly different probabilistic formalisms used by the input sources. One implication is that probability distributions can only be supplied as input data and not in the actual program. So far, input distributions are discrete and can only be given explicitly and not symbolically (e.g., Normal distribution with parameters mean and standard deviation).

- The probabilistic event language allows ENFrame to leverage existing work on incremental maintenance of query answers and extend it to incremental maintenance of the program output in the face of updates to input probabilities, insertions and deletions of uncertain objects (though this is subject to future work).

[1] Such events can be exponentially more succinct than equivalent propositional formulas over random variables or Bayesian networks.

```
1  (O, n) = loadData(db)      # list and number of objects
2  (k, iter) = loadParams()   # number of clusters/iterations
3  M = init()                 # initialise medoids
4
5  for it in range(0,iter):   # clustering iterations
6   InCl = [None] * k         # assignment phase
7   for i in range(0,k):
8    InCl[i] = [None] * n
9    for l in range(0,n):
10    InCl[i][l] = reduce_and(
11     [(dist(O[l],M[i]) <= dist(O[l],M[j]))
12       for j in range(0,k)])
13  InCl = breakTies1(InCl)   # each object in one cluster
14
15  DistSum = [None] * k      # update phase
16  for i in range(0,k):
17   DistSum[i] = [None] * n
18   for l in range(0,n):
19    DistSum[i][l] = reduce_sum(
20      [dist(O[l],O[p]) for p in range(0,n) if InCl[i][p]])
21
22  Centre = [None] * k
23  for i in range(0,k):
24   Centre[i] = [None] * n
25   for l in range(0,n):
26    Centre[i][l] = reduce_and(
27      [DistSum[i][l] <= DistSum[i][p] for p in range(0,n)])
28  Centre = breakTies2(Centre) # one medoid per cluster
29
30  M = [None] * k
31  for i in range(0,k):
32   M[i] = reduce_sum(
33      [O[l] for l in range(0,n) if Centre[i][l]])
```

$\forall i \text{ in } 0..n-1 : O^i \equiv \Phi(o_i) \otimes \vec{o}_i$
$M^0_{-1} \equiv \Phi(o_{\pi(0)}) \otimes \vec{o}_{\pi(0)}; \ldots; M^{k-1}_{-1}$
$\equiv \Phi(o_{\pi(k-1)}) \otimes \vec{o}_{\pi(k-1)}$

$\forall \text{it in } 0..\text{iter}-1 :$
$\forall i \text{ in } 0..k-1 :$
$\forall l \text{ in } 0..n-1 :$
$\mathsf{InCl}^{i,l}_{\mathsf{it}} \equiv \bigwedge_{j=0}^{k-1} [\mathsf{dist}(O^l, M^i_{\mathsf{it}-1}) \leq \mathsf{dist}(O^l, M^j_{\mathsf{it}-1})]$

// Encoding of `breakTies1` omitted

$\forall i \text{ in } 0..k-1 :$
$\forall l \text{ in } 0..n-1 :$
$\mathsf{DistSum}^{i,l}_{\mathsf{it}} \equiv \sum_{p=0}^{n-1} \mathsf{InCl}^{i,p}_{\mathsf{it}} \otimes \mathsf{dist}(O^l, O^p)$

$\forall i \text{ in } 0..k-1 :$
$\forall l \text{ in } 0..n-1 :$
$\mathsf{Centre}^{i,l}_{\mathsf{it}} \equiv \bigwedge_{p=0}^{n-1} [\mathsf{DistSum}^{i,l}_{\mathsf{it}} \leq \mathsf{DistSum}^{i,p}_{\mathsf{it}}]$

// Encoding of `breakTies2` omitted

$\forall i \text{ in } 0..k-1 :$
$M^i_{\mathsf{it}} = \sum_{l=0}^{n-1} \mathsf{Centre}^{i,l}_{\mathsf{it}} \wedge O^l$

Figure 1: k-medoids clustering specified as user program (left) and event program (right).

2 ENFrame by Example: k-medoids clustering

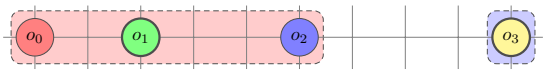
We illustrate ENFrame using the k-medoids algorithm, an iterative clustering algorithm with three phases. During the *initialisation phase*, k initial medoids (cluster centres) are selected at random or using a heuristic. Then, the algorithm iteratively assigns data points to the closest medoid during the *assignment phase*, followed by the *update phase* during which the new medoids are selected.

Figure 1 shows a user program in ENFrame encoding this algorithm and the corresponding event program.

The **user program** is written in a subset of Python and is oblivious to the uncertainty of the input data; consequently, the users need not be conversant with probabilistic data models and inference.

The data is loaded from a database (if a query and database are provided), or an external data source (line 1). The initial medoids are then selected (line 3). In the assignment phase (lines 6 – 13), for each cluster C_i and object o_l, the Boolean vector `InCl` stores whether o_l is assigned to C_i. The Boolean value `Incl[i][l]` is true if the distance from o_l to the medoid M_i of cluster C_i is the smallest of the distances to all medoids M_j (line 10). After the assignment phase, the update phase is performed to select the new medoids `M[i]` (lines 15 – 33). An object becomes a medoid if its distance to all other objects in the same cluster is the smallest of all cluster members. For each cluster and object, the sum of distances to other cluster members is computed and stored in `DistSum` (lines 15 – 20). For cluster C_i and object o_l, the Boolean vector `Centre` stores whether o_l is the new medoid of C_i. The value `Centre[i][l]` is true if the distance sum of o_i to members of C_i is the smallest of all objects in C_i.

Example 1: For $k = 2$ clusters and Euclidean distance, the user program can be run on the following deterministic data consisting of four data points on a line to yield the following result with o_1 and o_3 as cluster medoids:

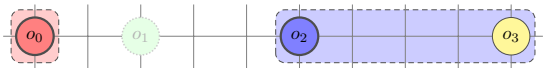


The **event program** gives a probabilistic interpretation of the user program: the deterministic variables in the user program become random variables. This gives rise to a probabilistic interpretation of k-medoids in which the object existence, cluster assignment, and medoid selection become uncertain.

ENFrame uses *conditional values* (c-values) to construct random variables whose sample space consists of real numbers and vectors in the feature space; the c-value construct is inspired by semimodule expressions used to capture provenance of queries with aggregates [2]. A c-value is denoted by $\Phi \otimes v$, where Φ is a propositional formula over Boolean random variables (or an arbitrary event) and v can be any real number or feature space vector. It is a probabilistic event whose interpretation is value v in case Φ is true and the neutral value for the domain of v otherwise (e.g., zero for reals); neutral values have a special interpretation in our formalism: An object whose value defaults to a neutral value in a possible world is interpreted as not existing in that world. Conditional values form the basic components of more complex events such as probabilistic sums, e.g., DistSum in the event program. Any discrete probability distribution $[(\vec{o}_1, p_1), \ldots, (\vec{o}_m, p_m)]$ over the positions $\vec{o}_1, \ldots, \vec{o}_m$ of a data point can be modelled using a sum of c-values $\sum_{j=1}^{m} \Phi_j \otimes \vec{o}_j$, where the Boolean probabilistic events $\Phi_1, \ldots, \Phi_m$ are mutually exclusive and have probabilities $p_1, \ldots, p_m$.

The random variables O_i are examples of c-values that define the input data points conditioned on probabilistic events. The Boolean random variable $\text{InCl}_{\text{it}}^{i,l}$ mirrors the Boolean variable `InCl[i][l]` in the user program; it represents the probabilistic event that object o_l is assigned to cluster C_i in iteration *it*. Similarly, $\text{DistSum}_{\text{it}}^{i,l}$ is a real-valued random variable representing all possible sums of distances from candidate medoid o_l to other objects in cluster C_i. Its probabilistic value is used to give rise to the Boolean random variable $\text{Centre}_{\text{it}}^{i,l}$, which eventually determines the new medoid M_{it}^{i}.

Example 2: Reconsider the clustering example from Example 1 and assume that the existence of each data point o_i is conditioned by a probabilistic event ϕ_i. Assume a total valuation ν of random variables, i.e., a possible world, that maps ϕ_1 to false and all other events ϕ_i to true. Then, the following result of 2-medoids clustering holds with a probability given by the product of the probabilities of the variable assignments in ν:



The program variables have a probabilistic interpretation and thus define a probability distribution that can be inspected by the user. For instance, the user can define a variable stating that two objects belong to the same cluster or co-occur together in clusters. ENFrame will compute the probability distributions for such variables. Such distributions can serve as input probabilistic data to queries and subsequent ENFrame programs.

3 Challenges Faced by ENFrame

ENFrame's goal is to enable probabilistic data programming beyond query processing. To achieve this goal, ENFrame faces several challenges, some of which are highlighted below.

Expressiveness of user language. Its user language can express popular data mining tasks mixed with queries. So far, we experimented with clustering (Markov, k-means, k-medoids), classification (k-nearest neighbour), and relational queries with aggregates. The user language supports bounded loops, conditional (if-then-else)

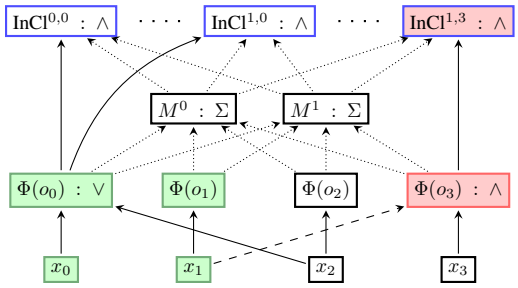


Figure 2: Excerpt of an event network for 2-medoids clustering with four data points $o_0, \ldots, o_3$.

statements, assignments, list comprehension and aggregation, primitive and array-valued variables. Using conditional statements and independent input random variables, any Bayesian network can be constructed in the user program. As exemplified for clustering, aggregations are key ingredients supported by ENFrame's language but missing in most existing probabilistic programming languages. The trade-off between the expressive power of the user language and the efficiency of probabilistic inference remains nevertheless an open research question.

For probabilistic inference, state-of-the-art approaches use Monte Carlo simulations that sample the input probability space, run the user program on each sample (possible world), and aggregate the results over the samples. With ENFrame, we seek to understand a complementary approach that is more in the spirit of earlier work on lineage-based query processing in probabilistic databases [20] and thus integrates well with the latter:

- (Grounding) We trace the lineage of the user program computation using probabilistic events, and then
- (Inference) We compute their exact probabilities or approximate them with error guarantees.

Each of these two tasks come with their own challenges beyond query processing[2].

Representation of event programs. Similarly to provenance management systems such as Orchestra [12] that represent provenance as graphs, ENFrame represents event programs as *event networks*, which are graphs whose nodes are Boolean connectives, comparisons, aggregates, and c-values. Expressions common to several events are only represented once in such graphs. Event networks for data mining tasks such as clustering are very repetitive and highly interconnected due to the combinatorial nature of the algorithms: The events at each iteration are expressions over the events at the previous iteration and have the same structure at each iteration. Moreover, the event networks can be cyclic, so as to account for program loops. While it is possible to unfold bounded-range loops, this can lead to prohibitively large event networks.

Example 3: Consider the event network in Figure 2. The lowest layer of the network consists of nodes for each of the random variables $x_0, \ldots, x_3$. On top of them are the events $\Phi(o_0), \ldots, \Phi(o_3)$ of the data points $o_0, \ldots, o_3$, which are $x_0 \vee x_2$, x_1, x_2, and $\neg x_1 \wedge x_3$ respectively. Furthermore, there are nodes for the medoids M^i and in-cluster Boolean values `InCl[i][l]`. Nodes in the lower layers in the network contribute to many (if not all) nodes in the next upper layer and sometimes even in higher layers. The dotted edges mean that nodes along them are not shown for readability reasons.

Inference via parallel approximate knowledge compilation. The inference techniques employed by ENFrame work for entire networks of interconnected events that are defined over several iterations, exhibit deep nesting structures, and use expressive constructs such as conditional values [21]. The common ground of these inference techniques is the compilation of the event networks into decision trees using Shannon expansion on the random variables occurring in the events. To scale to large networks and number of objects, we designed parallel approximate inference techniques. To approximate, we use an error budget to prune large, not-yet-explored

[2]Probabilistic query processing is a special instance of ENFrame processing: The lineage of a query on a probabilistic database is a probabilistic event that can be expressed in our event language and the probability of a query result is the probability of its lineage.

fragments of the decision tree that only account for a small probability mass. To parallelise the computation, we divide on-the-fly the decision tree into fragments for concurrent exploration.

Example 4: Consider again the network in Figure 2 and a partial assignment of the random variables that maps both x_0 and x_1 to true. The nodes turned green, i.e., x_0, x_1, $\Phi(o_0)$, and $\Phi(o_1)$, are then also mapped to true, while the red ones are mapped to false. Subsequent mappings of the remaining variables to true or false eventually leads to a truth value for the events of interest, which are the events sitting on top of the network.

Experiments with k-medoids clustering and k-nearest neighbour classification show that our exact inference technique performs orders of magnitude better than the naïve approach of iterating over all possible worlds. Furthermore, the approximate techniques (with absolute error guarantees) perform orders of magnitude better than exact inference. Parallel inference techniques scale almost linearly in the number of CPU cores (benchmarked on machines with 4-core and 16-core CPU). The experiments were conducted for settings of up to 15,000 objects and 150 input random variables, where the objects exhibit either positive, mutually exclusive, or conditional correlations. When using both parallelisation (16 cores) and approximation (absolute error of 0.1), ENFrame's performance is up to several seconds for the above settings. We plan to investigate the effect of employing several machines on performance of our inference techniques.

The output quality of our inference techniques has been evaluated against both the naïve method of iterating over all possible worlds (the golden standard), and inference in top-k most probable worlds. These experiments confirm that our error guarantees hold and show that top-k worlds inference only achieves the same output quality when k approaches the total number of worlds.

Managing large event networks. Full materialisation of event networks as obtained by exhaustive grounding of the user program can be prohibitive. We experimented with partial network compilation, where large network fragments are compiled down to C++ code. Initial experiments with k-medoids showed that this results in event networks that are two to three orders of magnitude smaller, while inference on the compressed event networks became up to an order of magnitude faster. All cluster membership events for a given cluster C_i can be lifted to one higher-order event, which expresses the membership of all data points to cluster C_i and can be compiled to C++ code for efficiency reasons. However, whereas the fine-grained events lead to large networks and less efficient inference, they support more accurately result explanation, sensitivity analysis, and incremental maintenance. We plan to investigate a functionality-performance trade-off brought by lifting the fine-grained events to their higher order realisation.

4 Conclusion

ENFrame is a programming framework for probabilistic data designed to integrate well with existing probabilistic database management systems such as MystiQ and MayBMS/SPROUT. So far, we investigated within this framework a rich language for random events that can trace complex computation such as clustering and classification, and distributed approximate inference techniques for networks of interconnected events with deep structure. Exciting directions for future work include:

- The trade-offs between the expressiveness of the user language and efficiency of inference and between the functionality of fine-grained events and performance brought by C++ compilation of event networks;
- Investigation of additional constructs in the event language needed to support a library of common algorithms for data analysis;
- Exploitation of the user program structure, in addition to query structure, for efficient inference;

- A better integration of queries and program constructs with relations and program data structures (e.g., multi-dimensional arrays) to be used interchangeably in both programs and queries;
- Improving the performance of inference by employing large-scale distributed systems.

References

[1] C. Aggarwal. *Managing and Mining Uncertain Data*. Kluwer, 2009.

[2] Y. Amsterdamer, D. Deutch, and V. Tannen. Provenance for aggregate queries. In *PODS*, pages 153–164, 2011.

[3] J. Boulos, N. N. Dalvi, B. Mandhani, S. Mathur, C. Ré, and D. Suciu. MYSTIQ: a system for finding more answers by using probabilities. In *SIGMOD*, pages 891–893, 2005.

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

[5] Defense Advanced Research Projects Agency. Probabilistic programming for advancing machine learning, April 2013. DARPA-BAA-13-31.

[6] X. L. Dong, E. Gabrilovich, G. Heitz, W. Horn, N. Lao, K. Murphy, T. Strohmann, S. Sun, and W. Zhang. Knowledge Vault: A web-scale approach to probabilistic knowledge fusion. In *SIGKDD*, 2014. To appear.

[7] R. Fink, L. Han, and D. Olteanu. Aggregation in probabilistic databases via knowledge compilation. *PVLDB*, 5(5):490–501, 2012.

[8] R. Fink, A. Hogue, D. Olteanu, and S. Rath. SPROUT^2: A squared query engine for uncertain web data. In *SIGMOD*, pages 1299–1302, 2011.

[9] R. Fink, J. Huang, and D. Olteanu. Anytime approximation in probabilistic databases. *VLDB J.*, pages 823–848, 2013.

[10] A. D. Gordon, T. A. Henzinger, A. V. Nori, and S. K. Rajamani. Probabilistic programming. In *ICSE*, 2014. Future of Software Engineering track (to appear).

[11] J. Huang, L. Antova, C. Koch, and D. Olteanu. MayBMS: a probabilistic database management system. In *SIGMOD*, pages 1071–1074, 2009.

[12] Z. Ives, T. Green, G. Karvounarakis, N. Taylor, V. Tannen, P. P. Talukdar, M. Jacob, and F. Pereira. The Orchestra collaborative data sharing system. *SIGMOD Rec.*, 37(2):26–32, 2008.

[13] R. Jampani, F. Xu, M. Wu, L. L. Perez, C. M. Jermaine, and P. J. Haas. MCDB: a monte carlo approach to managing uncertain data. In *SIGMOD*, pages 687–700, 2008.

[14] B. Kanagal, J. Li, and A. Deshpande. Sensitivity analysis and explanations for robust query evaluation in probabilistic databases. In *SIGMOD*, pages 841–852, 2011.

[15] B. Milch and et al. BLOG: Probabilistic models with unknown objects. In *IJCAI*, pages 1352–1359, 2005.

[16] D. Olteanu, J. Huang, and C. Koch. SPROUT: lazy vs. eager query plans for tuple-independent probabilistic databases. In *ICDE*, pages 640–651, 2009.

[17] D. Olteanu, L. Papageorgiou, and S. J. van Schaik. Πgora: An integration system for probabilistic data. In *ICDE*, pages 1324–1327, 2013.

[18] probabilistic-programming.org. Repository on probabilistic programming languages, 2014.

[19] P. Sen, A. Deshpande, and L. Getoor. PrDB: managing and exploiting rich correlations in probabilistic databases. *VLDB J.*, 18(5):1065–1090, 2009.

[20] D. Suciu, D. Olteanu, C. Ré, and C. Koch. *Probabilistic Databases*. Morgan & Claypool, 2011.

[21] S. J. van Schaik, D. Olteanu, and R. Fink. ENFrame: A platform for processing probabilistic data. In *EDBT*, pages 355–366, 2014.

[22] J. Widom. Trio: a system for data, uncertainty, and lineage. In C. Aggarwal, editor, *Managing and Mining Uncertain Data*, chapter 5. Springer-Verlag, 2008.

Feature Engineering for Knowledge Base Construction

Christopher Ré[†] Amir Abbas Sadeghian[†] Zifei Shan[†]
Jaeho Shin[†] Feiran Wang[†] Sen Wu[†] Ce Zhang[†‡]
[†]Stanford University
[‡]University of Wisconsin-Madison
{chrismre, amirabs, zifei, jaeho.shin, feiran, senwu, czhang}@cs.stanford.edu

Abstract

Knowledge base construction (KBC) is the process of populating a knowledge base, i.e., a relational database together with inference rules, with information extracted from documents and structured sources. KBC blurs the distinction between two traditional database problems, information extraction and information integration. For the last several years, our group has been building knowledge bases with scientific collaborators. Using our approach, we have built knowledge bases that have comparable and sometimes better quality than those constructed by human volunteers. In contrast to these knowledge bases, which took experts a decade or more human years to construct, many of our projects are constructed by a single graduate student.

Our approach to KBC is based on joint probabilistic inference and learning, but we do not see inference as either a panacea or a magic bullet: inference is a tool that allows us to be systematic in how we construct, debug, and improve the quality of such systems. In addition, inference allows us to construct these systems in a more loosely coupled way than traditional approaches. To support this idea, we have built the DeepDive system, which has the design goal of letting the user "think about features—not algorithms." We think of DeepDive as declarative in that one specifies what they want but not how to get it. We describe our approach with a focus on feature engineering, which we argue is an understudied problem relative to its importance to end-to-end quality.

1 Introduction

This document highlights what we believe is a critical and underexplored aspect of building high-quality knowledgebase construction (KBC) systems: ease of feature engineering. The hypothesis of our work is that the easier the feature engineering process is to debug and diagonse, the easier it is to improve the quality of the system. The single most important design goal of DeepDive is to make KBC systems easier to debug and improve. Although such techniques are important for end-to-end quality, techniques to debug KBC systems are understudied.

To describe our thoughts more precisely, we describe our motivation for building KBC systems, our choice of language that defines the syntax of our feature engineering problem, and a set of debugging techniques that we have found useful:[1]

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

[1]This document is superficial, and we refer the interested reader to more detailed material including example code and data that are available online. `http://deepdive.stanford.edu` contains the latest material and is under active construction.

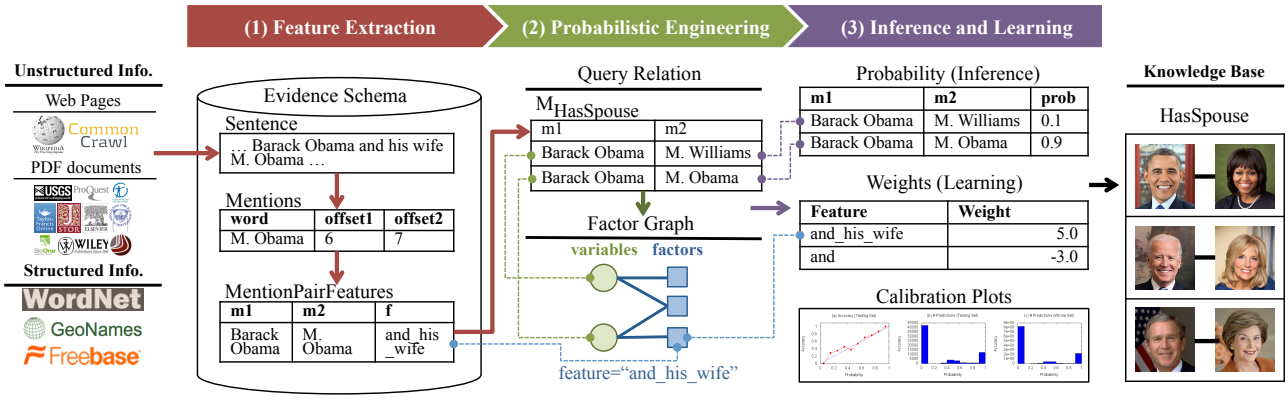


Figure 1: An overview of a KBC system built with DeepDive that takes as input both structured and unstructured information and creates a knowledge base as the output. There are three phases of DeepDive's execution model: (1) Feature Extraction; (2) Probabilistic Engineering; and (3) Inference and Learning. Section 2 contains a more detailed walkthrough of these phases, and Figure 4 shows more details of how to conduct tasks in these three phases using SQL and script languages, e.g., Python.

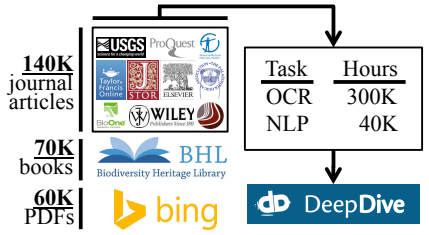
Motivation. In Section 2, we discuss a key motivating application for DeepDive, KBC. We take a holisitic approach that performs integration and extraction as a single task (rather than as two separate tasks). This holistic approach allows is to acquire data at all levels, including from larger and more diverse corpora, more human annotations, and/or larger (but still incomplete) knowledge bases. In turn, we can choose the source of data that has the best available signal for the least cost, a concept that we call *opportunistic acquisition*.

Language. In Section 2, we describe our choice of language model with the goal of abstracting the details of statistical learning and inference from the user to the extent possible. To integrate domain knowledge and support standard tools on top of DeepDive, we built our system on the relational model. Originally, we had our own custom-built language for inference based on Markov Logic [30], but we found that this language was unfamiliar to scientists. Instead, we decided to be as language agnostic as possible. A scientist can now write in almost any language to engineer features in DeepDive. However, for scalability reasons, bulk operations are written in SQL.

Debugging. In Section 4, we describe our approach to debugging. A key challenge is how to identify the relevant domain knowledge to integrate and what types of input sources have valuable signal. In our experience, without a systematic error analysis, it is not uncommon for developers to add rules that do not significantly improve the quality, which we call *prematurely optimizing* the KBC system. This is analogous to the popular engineering mistake of optimizing code without first profiling its end-to-end performance, akin to a failure to appreciate Amdahl's law. We view the key contribution of this work as highlighting the importance of error analysis in KBC applications.

Performance is also a major challenge. In our KBC systems using DeepDive, we may need to perform inference and learning on billions of highly correlated random variables. Therefore, one of our technical focus areas has been to speed up probabilistic inference [41, 40, 24, 23, 27]. Although this challenge is important, we do not focus on it in this paper, but there is much interesting work in this direction [5, 14, 4, 34, 31, 7] including work on how to automatically select algorithms from a declarative specification [17, 25, 25, 27]

(a) PaleoDeepDive Dataflow



(b) Quality Assessment

Relation	PDD # Ext.	PDD Acc.	Human Acc.
Taxon-Taxon	27 M	97%	92%
Taxon-Fm.	4 M	96%	84%
Fm.-Time	3 M	92%	89%
Fm.-Location	5 M	94%	90%

(c) Scientific Application: Biodiversity

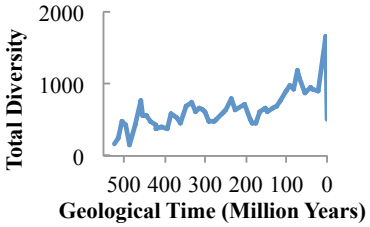


Figure 2: PaleoDeepDive, a KBC system for paleontology built with DeepDive. (a) The dataflow of PaleoDeepDive; (b) The assessment of quality for extractions in PaleoDeepDive (PDD); (c) One scientific application of PaleoDeepDive—biodiversity during the Phanerozoic period. More detailed results can be found in Peters et al. [28].

2 Knowledge Base Construction: The Case for Opportunistic Acquisition

Knowledge base construction is the process of populating a knowledge base with facts extracted from text, tabular data expressed in text and in structured forms, and even maps and figures. In *sample-based science* [28], one typically assembles a large number of facts (typically from the literature) to understand macroscopic questions, e.g., about the amount of carbon in the Earth's atmosphere throughout time, the rate of extinction of species, or all the drugs that interact with a particular gene. To answer such questions, a key step is to construct a high-quality knowledge base, and some forward-looking sciences have undertaken decade-long sample collection efforts, e.g., PaleoDB.org and PharmaGKB.org.

In parallel, KBC has attracted interest from industry [10, 42] and academia [18, 32, 33, 22, 9, 37, 2, 15, 3, 6, 29, 26]. To understand the common patters in KBC systems, we are actively collaborating with scientists from a diverse set of domains, including geology [38], paleontology [28], pharmacology for drug repurposing, and others. We briefly describe an application that we have constructed.

Example 1 (Paleontology and KBC [28]): Paleontology is based on the description and biological classification of fossils, an enterprise that has played out in countless collecting expeditions, museum visits, and an untold number of scientific publications over the past four centuries. One central task for paleontology is to construct a knowledge base about fossils from scientific publications, and an existing knowledge base compiled by human volunteers has greatly expanded the intellectual reach of paleontology and led to many fundamental new insights into macroevolutionary processes and the nature of biotic responses to global environmental change. However, the current process of using human volunteers is usually expensive and time-consuming. For example, PaleoDB, one of the largest such knowledge bases, took more than 300 professional paleontologists and 11 human years to build over the last two decades, resulting in `PaleoDB.org`. To get a sense of the impact of this database on this field, at the time of writing, this dataset has contributed to 205 publications, of which 17 have appeared in *Nature* or *Science*.

This provided an ideal test bed for our KBC research. In particular, we constructed a prototype called PaleoDeepDive [28] that takes in PDF documents. As a result, this prototype attacks challenges in optical character recognition, natural language processing, and information extraction and integration. Some statistics about the process are shown in Figure 2(a). As part of the validation of this system, we performed a double-blind experiment to assess the quality of the system versus the PaleoDB. We found that the KBC system built on DeepDive has achieved comparable—and sometimes better—quality than a knowledge base built by human volunteers over the last decade [28]. Figure 2(b) illustrates the accuracy of the results in PaleoDeepDive.

We have found that text is often not enough: often, the data that are interesting to scientists are located in

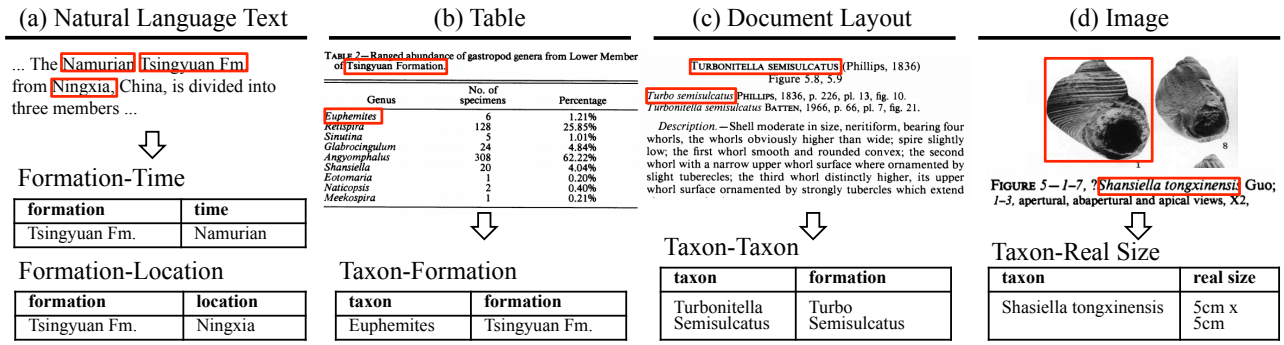


Figure 3: An illustration of data sources that PaleoDeepDive supports for KBC.

the tables and figures of articles. For example, in geology, more than 50% of the facts that we are interested in are buried in tables [11]. For paleontology, the relationship between taxa, as known as taxonomy, is almost exclusively expressed in section headers [28]. For pharmacology, it is not uncommon for a simple diagram to contain a large number of metabolic pathways. To build a KBC system with the quality that scientists will be satisfied with, we need to deal with these diverse sources of input. Additionally, external sources of information (other knowledge bases) typically contain high-quality signals (e.g., Freebase[2] and Macrostrat[3]). Leveraging these sources in information extraction is typically not studied in the classical information extraction context. To perform high-quality and high-coverage knowledge extraction, one needs a model that is able to ingest whatever presents itself *opportunistically*—that is, it is not tied solely to text but can handle more general extraction and integration.

Opportunistic Acquisition. We outline why we believe that the system must be able to *opportunistically acquire* data, by which we mean *the ability to acquire data from a wide variety of sources only to the degree to which they improve the end-to-end result quality*.

We describe the mechanisms by which we have improved the quality in KBC applications.[4] Consider one pattern that we observed in each of these KBC systems: we start from a state-of-the-art relation extraction model, e.g., a logistic regression model with rich linguistic features [21, 13]. Out of the box, such models do not have the quality that a domain expert requires. As a result, the expert must improve the model. This process of improving the model accounts for the dominant amount of time in which they interact with the system. To improve the extraction quality in terms of both precision and recall, the system needs some additional domain knowledge. For example, paleontologists might want to add a rule to improve the recall of a KBC system by informing the system about geological time; e.g., if a fossil appears in Late Cretaceous, then it also appears in the Cretaceous period. Biologists might want to add a rule to improve precision stating that one mention should only be recognized as a certain class of drug if there is clear linguistic clue that the drug is actually administered to a patient. These rules, however, must be expressed by domain scientists explicitly for the KBC system. Allowing users to express their knowledge is a challenge that has been studied in the extraction literature for some time, notably by SystemT [18, 19], which allows users to express their knowledge using declarative queries. In the next section, we describe our joint or collective approach, which is combined with a powerful technique called *distant supervision* that has allowed us to build knowledge bases with low cost. This technique allows DeepDive to take in this knowledge in many ways: rules, features, example datasets, and labeled data.

Choosing how to improve the system in the most effective way is arguably the key pain point in KBC. Nevertheless, we have noticed that there is a tendency to *prematurely optimize the quality* of intermediate results

[2] `http://www.freebase.com/`

[3] `http://macrostrat.org/`

[4] A complete list of the features we used for PaleoDeepDive can be found in our paper [28].

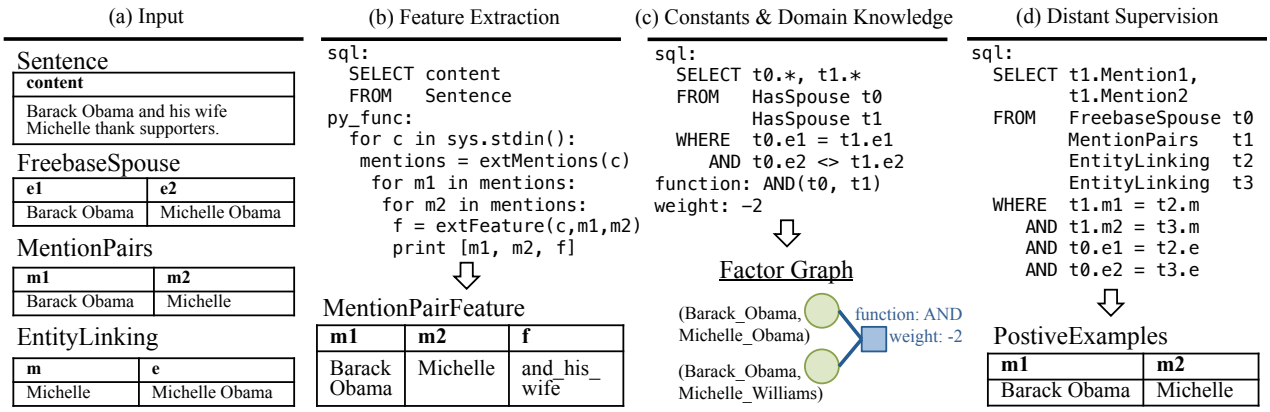


Figure 4: Illustration of popular operations in DeepDive. (a) Prepare data sets in relational form that can be used by DeepDive. (b) Generate labels using distant supervision with SQL; (c) Integrate constraints with SQL and logic functions; (d) Extract features with SQL and script languages (e.g., Python).

if the system is constructed in the absence of end-to-end goals. In our experience, we found that many of these premature optimizations seem reasonable in isolation and do locally improve the quality of the system. However, such optimizations have only marginal improvement on the end-to-end quality. As there is a never ending see of intermediate fixes prioritizing these fixes seems to be the critical issue. As a result, we advocate that one should prioritze how to improve the quality of the system based on how it affects the end-to-end quality of the system. We describe our first cut of how to make these choices in Section 4, with an emphasis on avoiding premature optimization.

3 A Brief Overview of DeepDive

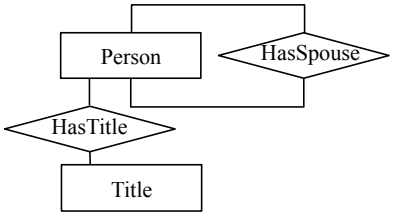
We briefly introduce the programming and execution model of DeepDive. Figure 4 shows the code that a user writes to interact with DeepDive. The reader can find a more detailed description of the language model of DeepDive in our previous work [26, 40] and on DeepDive's Web site. There are three phases in DeepDive's execution model, as shown in Figure 1:

(1) Feature Extraction. The input to this stage often contains both structured and unstructured information, and the goal is to produce a relational database that describes the features or signals of the data, which we call the *evidence schema*. We use the phrase evidence to emphasize that, in this stage, data is not required to be precisely correct as in traditional ETL; as a result, this is a much lighter ETL process.

- One responsibility of this phase is to run various OCR and NLP tools to acquire information from text, HTML, or images. The output database is often unnormalized: it may contain JSON objects to represent parse trees or DOM structures. This phase is essentially a high-throughput workflow system and may involve MapReduce jobs, Condor jobs, and SQL queries.
- The user also performs feature extraction in that they write user-defined functions (UDF) over existing query and evidence relations. DeepDive supports both ways, and the user can use SQL queries, and script languages, e.g., Python or Perl, to specify UDFs. Figure 4(b) and (d) show two examples.

(2) Probabilistic Engineering. The goal of this phase is to transform the evidence schema into a probabilistic model, specifically a factor graph that specifies:

(a) ER Diagram



(b) KBC Model

M_{Person}

id	Mention	Entity	offset
1	Michelle	Michelle_Obama	doc1_char1-8

M_{Title}

id	Mention	Entity	offset
4	Sen.	Senator	doc2_char5-8

$M_{HasSpouse}$

id	Mention1	Mention2
2	Barack Obama	Michelle

$M_{HasTitle}$

id	Mention1	Mention2
5	Obama	Sen.

HasSpouse

id	Entity1	Entity2
3	Barack_Obama	Michelle_Obama

HasTitle

id	Entity1	Entity2
6	Barack_Obama	Senator

Figure 5: Illustration of the KBC model in DeepDive. (a) An example ER diagram for the TAC-KBP application. (b) The corresponding KBC model, where each tuple in each relation corresponds to one random variable that DeepDive will predict. The column "offset" in mention tables M_{Person} and M_{Title} is the pointer to the position in the text, e.g., character offset in a document, and might be more complicated in real applications, e.g., a bounding box in a PDF.

- The set of random variables that the user wants to model. To define the set of random variables in DeepDive is to create new relations called query relations, in which each tuple corresponds to one random variable. This operation can be done by using SQL queries on existing relations.
- How those random variables are correlated; e.g., "The mention 'Obama' refers to the president is correlated with the random variable that indicates whether 'Obama' is a person." To specify *how*, the user specifies a factor function. Figure 4(c) shows a rough example that describes the intuition that *"people tend to be married to only a single person."* One (of many ways) to say this is to say that there is some correlation between pairs of married tuples; i.e., using the logical function AND(t0, t1) returns 1 in possible worlds in which both married tuples are true and 0 in others. DeepDive then learns the "strength" of this correlation from the data, which is encoded as weight.[5] Here, -2 indicates that it is less likely that both married tuples are correct. This phase is also where the user can write logical constraints, e.g., hard functional constraints.

Our previous work has shown that this grounding phase [23] can be a serious bottleneck if one does not use scalable relational technology. We have learned this lesson several times.

(3) Inference and Learning. This phase is largely opaque to the user: it takes the factor graph as input, estimates the weights, performs inference, and produces the output database along with a host of diagnostic information, notably calibration plots (see Fig. 6). More precisely, the output of DeepDive is a database that contains each random variable declared by the user with its marginal probability. For example, one tuple in the relation HasSpouse might be (Barack Obama, Michelle Obama), and ideally, we hope that DeepDive outputs a larger probability for this tuple as output.

3.1 Operations to Improve Quality in DeepDive

We illustrate the KBC model using one application called TAC-KBP,[6] in which the target knowledge base contains relations between persons, locations, and organizations. As is standard in the information extraction literature, a mention is a sequence of tokens in text, while an entity refers to the real-world object in the database. Figure 5(a) shows an excerpt from the ER diagram of the database. In this diagram, we have two types of entities, Person and Title, and two relations, HasSpouse and HasTitle. Figure 5(b) shows the KBC model induced

[5] A weight is roughly the log odds, i.e., the $\log \frac{p}{1-p}$ where p is the marginal probability of this random variable. This is standard in Markov Logic Networks [30], on which much of DeepDive's semantics are based.

[6] http://www.nist.gov/tac/2013/KBP/

from this ER diagram. For example, consider the Person entity and the HasSpouse relation. For Person, the KBC model contains a relation M_{Person}, which contains the candidate linking between a person mention (e.g., Michelle) to the person entity (e.g., Michelle_Obama or Michelle_Williams). The relation $M_{HasSpouse}$ contains mention pairs, which are candidates that participate in the HasSpouse relation, and the relation $HasSpouse$ contains the entity pairs. Each tuple of these relations corresponds to a Boolean random variable.

Routine Operations in DeepDive. We describe three routine tasks that a user performs to improve a KBC system.

An Example of Feature Extraction. The concept of a feature is one of the most important concepts for machine learning systems, and DeepDive's data model allows the user to use any scripting language for feature extraction. Figure 4(b) shows one such example using Python. One baseline feature that is often used in relation extraction systems is the word sequence between mention pairs in a sentence [21, 13], and Figure 4(b) shows an example of extracting this feature. The user first defines the input to the feature extractor using an SQL query, which selects all available sentences. Then the user defines a UDF that will be executed for each tuple returned by the SQL query. In this example, the UDF is a Python function that first reads a sentence from STDIN, extracts mentions from the sentence, extracts features for each mention pair, and outputs the result to STDOUT. DeepDive will then load the output of this UDF to the MentionPairFeature relation.

Constraints and Domain Knowledge. One way to improve a KBC system is to integrate domain knowledge, as we mentioned in Section 2. DeepDive supports this operation by allowing the user to integrate constraints and domain knowledge as correlations among random variables, as shown in Figure 4(c).

Imagine that the user wants to integrate a simple rule that says "one person is likely to be the spouse of only one person." For example, given a single entity "Barack_Obama," this rule gives positive preference to the case where only one of (Barack_Obama, Michelle_Obama) and (Barack_Obama, Michelle_Williams) is true. Figure 4(c) shows one example of implementing this rule. The SQL query in Figure 4(c) defines a view in which each tuple corresponds to two relation candidates with the same first entity but different second entities. The function AND(t0, t1) defines the "type of correlation" among variables, and the weight "-2" defines the strength of the correlation. This rule indicates that the it is less likely that both (Barack_Obama, Michelle_Obama) and (Barack_Obama, Michelle_Williams) are true (i.e., when AND(t0, t1) returns 1). Typically, DeepDive is used to learn the weights from data.

Distant Supervision. One challenge with building a machine learning system for KBC is generating training examples. As the number of predicates in the system grows, specifying training examples for each relation is tedious and expensive. One common technique to cope with this is distant supervision. Distant supervision starts with an (incomplete) entity-level knowledge base and a corpus of text. The user then defines a (heuristic) mapping between entities in the database and text. This map is used to generate (noisy) training data for mention-level relations [21, 13]. We illustrate this procedure by example.

Example 2 (Distant Supervision): Consider the mention-level relation $M_{HasSpouse}$. To find training data, we find sentences that contain mentions of pairs of entities that are married, and we consider the resulting sentences positive examples. Negative examples could be generated from pairs of persons who are, say, parent-child pairs. Ideally, the patterns that occur between pairs of mentions corresponding to mentions will contain indicators of marriage more often than those that are parent-child pairs (or other pairs). Selecting those indicative phrases or features allows us to find features for these relations and generate training data. Of course, engineering this mapping is a difficult task in practice and requires many iterations.

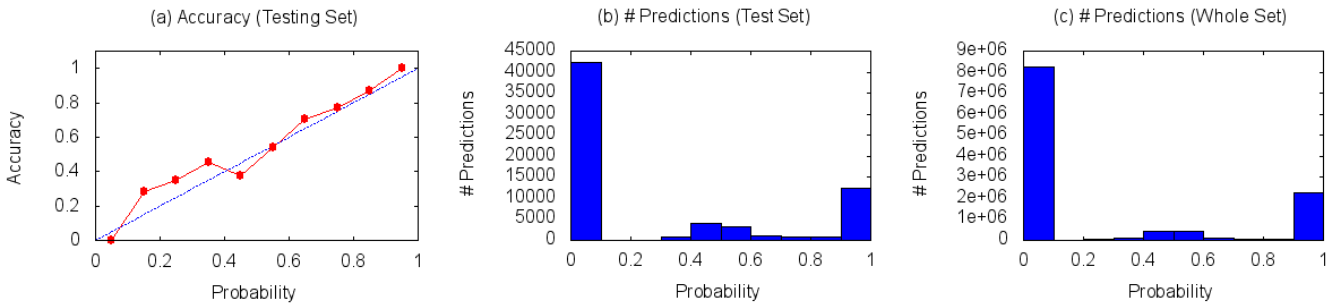


Figure 6: Illustration of calibration plots in DeepDive.

Concretely, Figure 4(d) shows an example of distant supervision in DeepDive. The FreebaseSpouse relation is an entity-level relation in Freebase (it contains pairs of married people). The EntityLinking relation specifies a mapping from entities to mentions in the text. The user provides an SQL query like the one shown in Figure 4(d) to produce another relation, PositiveExamples, that contains mention-level positive training examples. In our applications, users may spend time improving this mapping, which can lead to higher quality (but imperfect) training data much more cheaply than having a human label this data.

As we have described, the user has at least the above three ways to improve the system and is free to use one or a combination of them to improve the system's quality. The question we address next is, *"What should the user do next to get the largest quality improvement in the KBC system?"*

4 Debugging and Improving a KBC System

A DeepDive system is only as good as its features and rules. In the last two years, we have found that understanding which features to add is the most critical—but often the most overlooked—step in the process. Without a systematic analysis of the errors of the system, developers often add rules that do not significantly improve their KBC system, and they settle for suboptimal quality. In this section, we describe our process of error analysis, which we decompose into two stages: a macro-error analysis that is used to guard against statistical errors and gives an at-a-glance description of the system and a fine-grained error analysis that actually results in new features and code being added to the system.

4.1 Macro Error Analysis: Calibration Plots

In DeepDive, *calibration plots* are used to summarize the overall quality of the KBC results. Because DeepDive uses a joint probability model, each random variable is assigned a marginal probability. Ideally, if one takes all the facts to which DeepDive assigns a probability score of 0.95, then 95% of these facts are correct. We believe that probabilities remove a key element: the developer reasons about features, not the algorithms underneath. This is a type of *algorithm independence* that we believe is critical.

DeepDive programs define one or more test sets for each relation, which are essentially a set of labeled data for that particular relation. This set is used to produce a calibration plot. Figure 6 shows an example calibration plot for the Formation-Time relation in PaleoDeepDive, which provides an aggregated view of how the KBC system behaves. By reading each of the subplots, we can get a rough assessment of the next step to improve our KBC system. We explain each component below.

As shown in Figure 6, a calibration plot contains three components: (a) accuracy, (b) # predictions (test set), which measures the number of extractions in the test set with a certain probability; and (c) # predictions

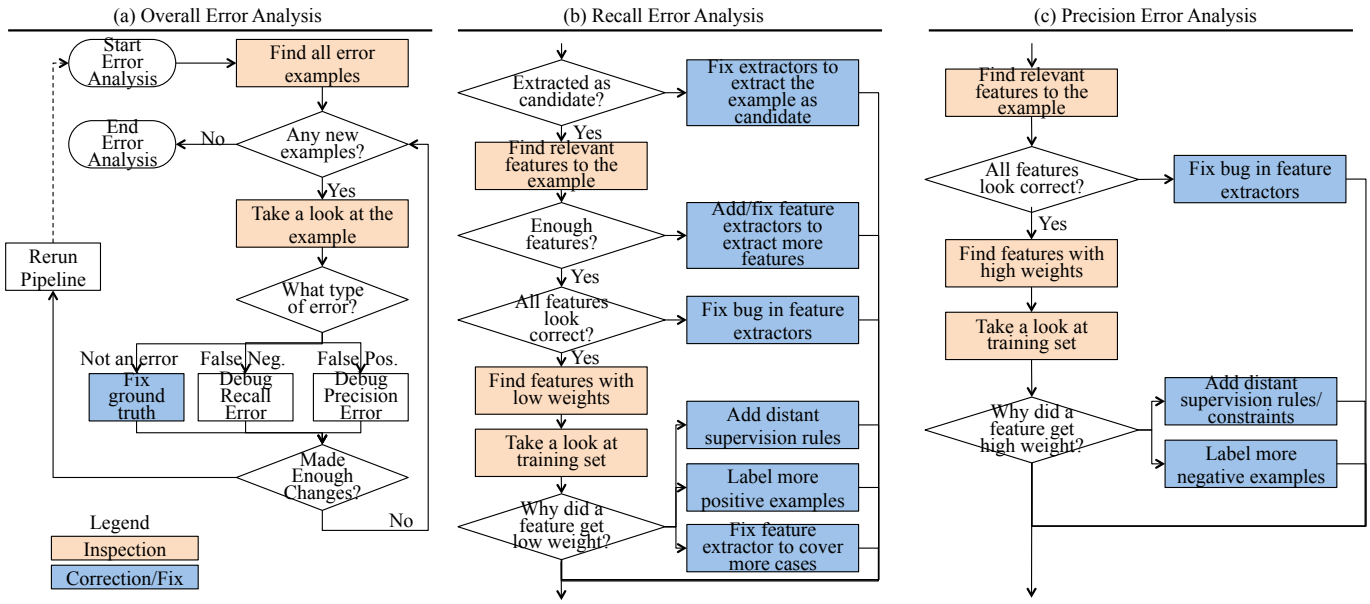


Figure 7: Error Analysis Workflow of DeepDive.

(whole set), which measures the number of extractions in the whole set with a certain probability. The test set is assumed to have labels so that we can measure accuracy, while the whole set does not.

(a) Accuracy. To create the accuracy histogram, we bin each fact extracted by DeepDive on the test set by the probability score assigned to each fact; e.g., we round to the nearest value in the set $k/10$ for $k = 1, \ldots, 10$. For each bin, we compute the fraction of those predictions that is correct. Ideally, this line would be on the (0,0)-(1,1) line, which means that the DeepDive-produced probability value is calibrated, i.e., it matches the *test-set accuracy*. For example, Figure 6(a) shows a curve for calibration. Differences in these two lines can be caused by noise in the training data, quantization error due to binning, or sparsity in the training data.

(b) # Predictions (Testing Set). We also create a histogram of the number of predictions in each bin. In a well-tuned system, the # Predictions histogram should have a "U" shape. That is, most of the extractions are concentrated at high probability and low probability. We do want a number of low-probability events, as this indicates DeepDive is considering plausible but ultimately incorrect alternatives. Figure 6(b) shows a U-shaped curve with some masses around 0.5-0.6. Intuitively, this suggests that there is some hidden type of example for which the system has insufficient features. More generally, facts that fall into bins that are not in (0,0.1) or (0.9,1.0) are candidates for improvements, and one goal of improving a KBC system is to "push" these probabilities into either (0,0.1) or (0.9,1.0). To do this, we may want to sample from these examples and add more features to resolve this uncertainty.

(c) # Predictions (Whole Set). The final histogram is similar to Figure 6(b), but illustrates the behavior of the system, for which we do not have any training examples. We can visually inspect that Figure 6(c) has a similar shape to (b); If not, this would suggest possible overfitting or some bias in the selection of the hold-out set.

4.2 Micro Error Analysis: Per-Example Analysis

Calibration plots provide a high-level summary of the quality of a KBC system, from which developers can get an abstract sense of how to improve it. For developers to produce rules that can improve the system, they need to actually look at data. We describe this fine-grained error analysis, illustrated in Figure 7.

Figure 7(a) illustrates the overall workflow of error analysis. The process usually starts with the developer finding a set of errors, which is a set of random variables having predictions that are inconsistent with the training data. Then, for each random variable, the developer looks at the corresponding tuples in the user relation (recall that each tuple in the user relation corresponds to a random variable) and tries to identify the types of errors. We describe two broad classes of errors that we think of as improving the recall and the precision, which we describe below.

Recall Error Analysis. A recall error, i.e., a false negative error, occurs when DeepDive is expected to extract a fact but fails to do so. In DeepDive's data model, this usually means that a random variable has a low probability or that the corresponding fact is not extracted as a candidate. Figure 7(b) illustrates the process of debugging a recall error.

First, it is possible that the corresponding fact that is expected to be extracted does not appear as a candidate. In this case, there is no random variable associated with this fact, so it is impossible for DeepDive to extract it. For example, this case might happen when the corresponding fact appears in tables, but we only have a text-based extractor. In this case, the developer needs to write extractors to extract these facts as candidates.

Another common scenario is when the corresponding fact appears as a candidate but is not assigned a high enough probability by DeepDive. In this case, the developer needs to take a series of steps, as shown in Figure 7(b), to correct the error. First, it is possible that this candidate is not associated with any features or that the corresponding features have an obvious bug. In this case, one needs to fix the extractors of the features. Second, it is possible that the corresponding features have a relatively low weight learned by DeepDive. In that case, one needs to look at the training data to understand the reason. For example, one might want to add more distant supervision rules to increase the number of positive examples that share the same feature or even manually label more positive examples. One might also want to change the feature extractors to collapse the current features with other features to reduce the sparsity.

Precision Error Analysis. A precision error, i.e., a false positive error, occurs when DeepDive outputs a high probability for a fact, but it is an incorrect one or not supported by the corresponding document. Figure 7(c) illustrates the process of debugging a precision error. Similar to debugging recall errors, for precision errors, the developer needs to look at the features associated with the random variable. The central question to investigate is why some of these features happen to have high weights. In our experience, this usually means that we do not have enough negative examples for training. If this is the case, the developer adds more distant supervision rules or (less commonly) manually labels more negative examples.

Avoiding Overfitting. One possible pitfall for the error analysis at the per-example level is overfitting, in which the rules written by the developer overfit to the corpus or the errors on which the error analysis is conducted. To avoid this problem, we have insisted in all our applications on conducting careful held-out sets to produce at least (1) a training set; (2) a testing set; and (3) an error analysis set. In the error analysis process, the developer is only allowed to look at examples in the error analysis set, and validate the score on the testing set.

5 Related Work

Knowledge Base Construction (KBC) has been an area of intense study over the last decade [18, 32, 33, 22, 9, 37, 2, 15, 3, 6, 42, 29]. Within this space, there are a number of approaches.

Rule-Based Systems. The earliest KBC systems used pattern matching to extract relationships from text. The most well-known example is the "Hearst Pattern" proposed by Hearst [12] in 1992. In her seminal work, Hearst observed that a large number of hyponyms can be discovered by simple patterns, e.g., "X such as Y." Hearst's technique has formed the basis of many further techniques that attempt to extract high-quality patterns from text. Rule-based (pattern matching-based) KBC systems, such as IBM's SystemT [18, 19], have been built to aid developers in constructing high-quality patterns. These systems provide the user with a (declarative) interface to specify a set of rules and patterns to derive relationships. These systems have achieved state-of-the-art quality on tasks such as parsing [19].

Statistical Approaches. One limitation of rule-based systems is that the developer needs to ensure that all rules provided to the system are high-precision rules. For the last decade, probabilistic (or machine learning) approaches have been proposed to allow the system to select from a range of a priori features automatically. In these approaches, the extracted tuple is associated with a marginal probability that it is true. DeepDive, Google's knowledge graph, and IBM's Watson are built on this approach. Within this space, there are three styles of systems:

- **Classification-Based Frameworks.** Here, traditional classifiers assign each tuple a probability score, e.g., a naïve Bayes classifier or a logistic regression classifier. For example, KnowItAll [9] and TextRunner [37, 2] use a naïve Bayes classifier, and CMUs NELL [3, 6] uses logistic regression. Large-scale systems typically use these types of approaches in sophisticated combinations, e.g., NELL or Watson.
- **Maximum a Posteriori (MAP).** Here, a probabilistic approach is used, but the MAP or most likely world (which do differ slightly) is selected. Notable examples include the YAGO system [15], which uses a PageRank-based approach to assign a confidence score. Other examples include SOFIE [33] and Prospera [22], which use an approach based on constraint satisfaction.
- **Graphical Model Approaches.** The classification-based methods ignore the interaction among predictions, and there is a hypothesis that modeling these correlations yields higher quality systems more quickly. A generic graphical model has been used to model the probabilistic distribution among all possible extractions. For example, Poon et al. [29] used Markov logic networks (MLN) [8] for information extraction. Microsoft's StatisticalSnowBall/EntityCube [42] also uses an MLN-based approach. A key challenge in these systems is scalability. For example, Poon et al. was limited to 1.5K citations. Our relational database-driven algorithms for MLN-based systems are dramatically more scalable [23].

6 Future Work

DeepDive is our first step toward facilitating the building of knowledge base construction systems of sufficient quality to support applications, including rigorous scientific discoveries. Given our experience in interacting with scientists and understanding their needs, we found the following directions interesting candidates for future exploration, most of which seem to be challenging open research problems.

- **Dealing with Time and Temporal Information.** The KBC model we describe in this work treats time as a distinguished predicate such that each fact is extracted as long as there exists a time point such that it is true. However, prior work has shown that each fact could be associated with a more expressive temporal

tag [20, 36, 35], and our previous participation in the DARPA machine reading challenge also attempted to produce a KBC system with such tags. From our conversations with scientists, we have found that these temporal tags are sometimes important. For example, in paleontology, where the corpus spans more than 400 years, the ability to find the most recent version of the fact is important to produce the up-to-date view of the tree of life. It is interesting to study how to integrate temporal information into DeepDive's KBC model and conduct reasoning over it.

- **Visual Information Extraction.** As we mentioned in Section 2, we observed that there is a vast amount of information buried in data sources such as images and diagrams. Although the current DeepDive system can perform some simple image-based extraction, e.g., to extract body size from paleontology publications [28], it is an interesting direction to study how to extract information from tables, charts, and diagrams.
- **Incremental Processing.** One observation we have about debugging a KBC system is that, to conduct error-analysis efficiently, one usually needs to rerun the system with slightly different DeepDive programs (e.g., more rules) and data (e.g., more structural resources). In our application, it is not uncommon for the factor graph to contain billions of random variables. One obvious solution to this is to study how to incrementally conduct inference and learning to support more efficient error analysis. This problem is distinct from incremental or online learning, as the focus is on maintaining the downstream data products. We have done some work in this direction in a simplified classifier-based setting [16].
- **"Active Debugging" and Rule Learning.** The current debugging infrastructure in DeepDive is "passive" in that DeepDive waits for instructions from the user for error analysis or adding more features. One future direction is to study whether DeepDive can play a more "active" role. One example is whether DeepDive could automatically suggest errors for the users to investigate or report relations where more training data are required. It is also interesting to study whether it is possible for DeepDive to automatically learn inference rules or feature extractors and recommend them to the user. We envisioned a similar system in our previous work [1] and have worked on a domain-specific language for a related problem [39].
- **Even-More-joint Information Extraction.** As we show in this work, one of DeepDive's advantage is the ability to jointly take advantage of different sources of signals. One interesting question is whether we can extend the current KBC model to be even more joint. First, it is interesting to study how to use DeepDive for low-level NLP tasks, e.g., linguistic parsing and optical character recognition, and how these low-level tasks interact with high-level tasks, e.g., relation extraction. Second, it is interesting to study how application-level knowledge, e.g., the R-script used by paleontologists over DeepDive's knowledge base, can be used to constrain and improve the extraction task.
- **Visualization of Uncertainty.** The current DeepDive system expresses the uncertainty of inference results to the user using marginal probabilities. Even simple visualizations have proved to be far more effective at conveying debugging information about statistical or aggregate properties.

7 Conclusion

We have described what we believe is one of the key features of DeepDive: the ability to rapidly debug and engineer a better system. We have argued that probabilistic and machine learning techniques are critical, but only in that they enable developers to think in terms of features—not algorithms.

8 Acknowledgments

We would like to thank the users of DeepDive, especially Shanan Peters and Emily Doughty, who have given a great deal of helpful (and patient) feedback. We would also like to thank Michael J. Cafarella and Dan Suicu

who gave comments on an earlier draft of this document. We gratefully acknowledge the support of the Defense Advanced Research Projects Agency (DARPA) XDATA Program under No. FA8750-12-2-0335 and DEFT Program under No. FA8750-13-2-0039, DARPA's MEMEX program, the National Science Foundation (NSF) CAREER Award under No. IIS-1353606 and EarthCube Award under No. ACI-1343760, the Office of Naval Research (ONR) under awards No. N000141210041 and No. N000141310129, the Sloan Research Fellowship, American Family Insurance, Google, and Toshiba. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DARPA, AFRL, NSF, ONR, or the U.S. government.

References

[1] M. Anderson, D. Antenucci, V. Bittorf, M. Burgess, M. J. Cafarella, A. Kumar, F. Niu, Y. Park, C. Ré, and C. Zhang. Brainwash: A data system for feature engineering. In *CIDR*, 2013.

[2] M. Banko, M. J. Cafarella, S. Soderland, M. Broadhead, and O. Etzioni. Open information extraction from the Web. In *IJCAI*, 2007.

[3] J. Betteridge, A. Carlson, S. A. Hong, E. R. Hruschka, E. L. M. Law, T. M. Mitchell, and S. H. Wang. Toward never ending language learning. In *AAAI Spring Symposium*, 2009.

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

[5] J. Canny and H. Zhao. Big Data analytics with small footprint: Squaring the cloud. In *KDD*, 2013.

[6] A. Carlson, J. Betteridge, B. Kisiel, B. Settles, E. R. Hruschka, and T. M. Mitchell. Toward an architecture for never-ending language learning. In *AAAI*, 2010.

[7] Y. Chen and D. Z. Wang. Knowledge expansion over probabilistic knowledge bases. In *SIGMOD*, 2014.

[8] P. Domingos and D. Lowd. *Markov Logic: An Interface Layer for Artificial Intelligence*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2009.

[9] O. Etzioni, M. Cafarella, D. Downey, S. Kok, A.-M. Popescu, T. Shaked, S. Soderland, D. S. Weld, and A. Yates. Web-scale information extraction in KnowItAll: (preliminary results). In *WWW*, 2004.

[10] D. A. Ferrucci, E. W. Brown, J. Chu-Carroll, J. Fan, D. Gondek, A. Kalyanpur, A. Lally, J. W. Murdock, E. Nyberg, J. M. Prager, N. Schlaefer, and C. A. Welty. Building Watson: An overview of the DeepQA project. *AI Magazine*, 2010.

[11] V. Govindaraju, C. Zhang, and C. Ré. Understanding tables in context using standard NLP toolkits. In *ACL*, 2013.

[12] M. A. Hearst. Automatic acquisition of hyponyms from large text corpora. In *COLING*, 1992.

[13] R. Hoffmann, C. Zhang, and D. S. Weld. Learning 5000 relational extractors. In *ACL*, 2010.

[14] R. Jampani, F. Xu, M. Wu, L. L. Perez, C. Jermaine, and P. J. Haas. MCDB: A Monte Carlo approach to managing uncertain data. In *SIGMOD*, 2008.

[15] G. Kasneci, M. Ramanath, F. Suchanek, and G. Weikum. The YAGO-NAGA approach to knowledge discovery. *SIGMOD Rec.*, 2009.

[16] M. L. Koc and C. Ré. Incrementally maintaining classification using an RDBMS. *PVLDB*, 2011.

[17] T. Kraska, A. Talwalkar, J. C. Duchi, R. Griffith, M. J. Franklin, and M. I. Jordan. Mlbase: A distributed machine-learning system. In *CIDR*, 2013.

[18] R. Krishnamurthy, Y. Li, S. Raghavan, F. Reiss, S. Vaithyanathan, and H. Zhu. SystemT: A system for declarative information extraction. *SIGMOD Rec.*, 2009.

[19] Y. Li, F. R. Reiss, and L. Chiticariu. SystemT: A declarative information extraction system. In *HLT*, 2011.

[20] X. Ling and D. S. Weld. Temporal information extraction. In *AAAI*, 2010.

[21] M. Mintz, S. Bills, R. Snow, and D. Jurafsky. Distant supervision for relation extraction without labeled data. In *ACL*, 2009.

[22] N. Nakashole, M. Theobald, and G. Weikum. Scalable knowledge harvesting with high precision and high recall. In *WSDM*, 2011.

[23] F. Niu, C. Ré, A. Doan, and J. Shavlik. Tuffy: Scaling up statistical inference in Markov logic networks using an RDBMS. *PVLDB*, 2011.

[24] F. Niu, B. Recht, C. Ré, and S. J. Wright. Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. In *NIPS*, 2011.

[25] F. Niu, C. Zhang, C. Ré, and J. W. Shavlik. Felix: Scaling inference for Markov Logic with an operator-based approach. *ArXiv e-print.*, 2011.

[26] F. Niu, C. Zhang, C. Ré, and J. W. Shavlik. Elementary: Large-scale knowledge-base construction via machine learning and statistical inference. *Int. J. Semantic Web Inf. Syst.*, 2012.

[27] F. Niu, C. Zhang, C. Ré, and J. W. Shavlik. Scaling inference for Markov logic via dual decomposition. In *ICDM*, 2012.

[28] S. E. Peters, C. Zhang, M. Livny, and C. Ré. A machine-compiled macroevolutionary history of Phanerozoic life. *ArXiv e-prints*, 2014.

[29] H. Poon and P. Domingos. Joint inference in information extraction. In *AAAI*, 2007.

[30] M. Richardson and P. Domingos. Markov logic networks. *Machine Learning*, 2006.

[31] P. Sen, A. Deshpande, and L. Getoor. PrDB: Managing and exploiting rich correlations in probabilistic databases. *The VLDB Journal*, 2009.

[32] W. Shen, A. Doan, J. F. Naughton, and R. Ramakrishnan. Declarative information extraction using datalog with embedded extraction predicates. In *VLDB*, 2007.

[33] F. M. Suchanek, M. Sozio, and G. Weikum. SOFIE: A self-organizing framework for information extraction. In *WWW*, 2009.

[34] D. Z. Wang, M. J. Franklin, M. Garofalakis, J. M. Hellerstein, and M. L. Wick. Hybrid in-database inference for declarative information extraction. In *SIGMOD*, 2011.

[35] Y. Wang, B. Yang, S. Zoupanos, M. Spaniol, and G. Weikum. Scalable spatio-temporal knowledge harvesting. In *WWW*, 2011.

[36] Y. Wang, M. Zhu, L. Qu, M. Spaniol, and G. Weikum. Timely YAGO: harvesting, querying, and visualizing temporal knowledge from Wikipedia. In *EDBT*, 2010.

[37] A. Yates, M. Cafarella, M. Banko, O. Etzioni, M. Broadhead, and S. Soderland. TextRunner: Open information extraction on the Web. In *NAACL*, 2007.

[38] C. Zhang, V. Govindaraju, J. Borchardt, T. Foltz, C. Ré, and S. Peters. GeoDeepDive: statistical inference using familiar data-processing languages. In *SIGMOD*, 2013.

[39] C. Zhang, A. Kumar, and C. Ré. Materialization optimizations for feature selection workloads. In *SIGMOD*, 2014.

[40] C. Zhang and C. Ré. Towards high-throughput Gibbs sampling at scale: a study across storage managers. In *SIGMOD*, 2013.

[41] C. Zhang and C. Ré. DimmWitted: A study of main-memory statistical analytics. *PVLDB*, 2014.

[42] J. Zhu, Z. Nie, X. Liu, B. Zhang, and J.-R. Wen. StatSnowball: A statistical approach to extracting entity relationships. In *WWW*, 2009.

Efficient In-Database Analytics with Graphical Models

Daisy Zhe Wang, Yang Chen, Christan Grant and Kun Li
{daisyw,yang,cgrant,kli}@cise.ufl.edu
University of Florida, Department of Computer and Information Science and Engineering

Abstract

Due to recent application push, there is high demand in industry to extend database systems to perform efficient and scalable in-database analytics based on probabilistic graphical models (PGMs). We discuss issues in supporting in-database PGM methods and present techniques to achieve a deep integration of the PGM methods into the relational data model as well as the query processing and optimization engine. This is an active research area and the techniques discussed are being further developed and evaluated.

1 Introduction

Graphical models, also known as probabilistic graphical models (PGMs), are a class of expressive and widely adopted machine learning models that compactly represent the joint probability distribution over a large number of interdependent random variables. They are used extensively in large-scale data analytics, including information extraction (IE), knowledge base population (KBP), speech recognition and computer vision. In the past decade, many of these applications have witnessed dramatic increase in data volume. As a result, PGM-based data analytics need to scale to terabytes of data or billions of data items. While new data-parallel and graph-parallel processing platforms such as Hadoop, Spark and GraphLab [1, 36, 3] are widely used to support scalable PGM methods, such solutions are not effective for applications with data residing in databases or with need to support online PGM-based data analytics queries.

More specifically, large volumes of valuable data are likely to pour into database systems for many years to come due to the maturity of the commercial database systems with transaction support and ACID properties and due to the legal auditing and data privacy requirements in many organizations. Examples include electronic medical records (EMRs) in EPIC systems as well as financial transactions in banks and credit card companies. Unfortunately, lacking native in-database analytics support, these volumes of data have to be transferred in and out of the database for processing, which is of potentially huge overhead. It would be much more efficient to push analytic computation close to data.

Moreover, efficient support of *online queries* are required by many applications for interactive data analytics. Most parallel data processing frameworks only support off-line batch-oriented analytics, which is very in-efficient for online PGM-based analytics driven by ad-hoc queries. In these cases, PGM-based data analytics methods should be pushed into database systems as first-class citizens in the data model, query processing and query optimization. To achieve efficiency and scalability, a deep integration of graphical models and algorithms with a database system is essential.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

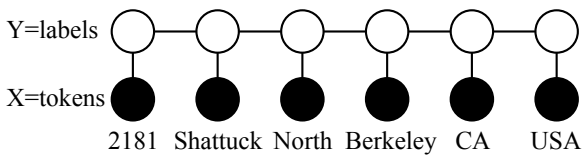
As a result, there is a need to extend database systems to perform large-scale in-database analytics based on probabilistic graphical models. Such an extension requires first-class modeling and implementation of probabilistic graphical models and algorithms to support query-time model-based inference and reasoning. New query optimization and view maintenance techniques are also needed to support queries with both relational and statistical analytics operations. The rest of the article describes our efforts to address three major challenges in supporting PGMs efficiently in a database system:

- *Model Representation and Model-Data Join*: First, we describe a relational representation of graphical models and model-data join algorithms that grounds a first-order PGM over a large number of data instances, resulting in a large propositional PGM [4].
- *Efficient In-Database Statistical Inference*: Second, we describe mechanisms that are required for efficient implementation of inference operations in a database system, including data-driven feature extraction and iterative inference computations over grounded PGMs [5].
- *Optimizing Queries with Inference*: Third, we describe new query optimization techniques to support online queries involving PGM-based analytics, which contain relational and inference operators over graphical models [6, 7].

Related Work Database systems and technologies lack the facilities to store probabilistic graphical models and perform statistical learning and inference tasks over PGMs. Three bodies of recent work have made progress in supporting efficient and scalable PGM methods in database for advanced data analytics. First, there are recent work on supporting in-database statistical analytics and machine learning methods such as linear algebra, matrix manipulation and convex optimization [8, 5, 9]. Second, graphical models and algorithms are used in probabilistic database literature to represent and manipulate uncertain data with high-dimensional joint distributions [10, 11, 12]. Third, recent work show that some algorithms, such as Viterbi, sum-product and Markov chain Monte Carlo (MCMC) sampling, commonly used for inference over graphical models can be efficiently supported and optimized in a relational query processing database system [13, 14, 15, 7].

2 Probabilistic Graphical Models

Probabilistic graphical models use a graph-based representation as the basis for compactly encoding a complex distribution over a high-dimensional space [16]. In this graphical representation, the nodes correspond to the variables in our domain, and the edges correspond to direct probabilistic interactions between them. As an example, Figure 1(a) shows a graphic model for text analysis.



(a)

X_1	Y_1	$\Phi(X_1, Y_1)$
2181	street number	22
2181	street name	5
2181	zip code	8
...	...	...

(b)

Figure 1: (a) An example linear-chain CRF model. (b) Factor $\Phi(X_1, Y_1)$ showing the probabilistic correlation between variables X_1 and Y_1.

The variables are an observed token sequence $\mathbf{X}$ and its label sequence $\mathbf{Y}$ to be predicted. The labels can be a street number, a city, a zip code, etc. The edges represent the probabilistic correlations among the variables, defined by *factors*. In this example, there are two types of edges. The vertical edges represent the factors between token and corresponding label. For instance, token "2181" is more likely to be a street number than a street name or a zip code. Hence, we have a factor in Figure 1(b) to reflect this correlation. As shown in the table, the values of $\Phi(X_1, Y_1)$ can be any real numbers that indicate the relevant likelihood of different assignments and do not

need to be probabilities. Likewise, the horizontal edges represent factors between adjacent labels. For instance, if the previous label is a street number, then a street name is likely to follow.

Together, these factors define a probability distribution $P(\mathbf{Y}|\mathbf{X})$ that can be used to make label predictions. This particular kind of linear graphical models for text analysis is studied extensively and is called linear-chain *conditional random field* (CRF) [17]. The task of computing or approximating the probability distribution $P(\mathbf{Y}|\mathbf{X})$ is called *inference*. Hence, inference is a key step toward predicting the labels $\mathbf{Y}$.

More recently, statistical relational learning (SRL) models combine PGMs with first-order logic to model the uncertainty and probabilistic correlations in relational domains. SRL models can be considered as first-order PGMs, which are *grounded* into propositional PGMs at inference time. As an example, Markov logic networks (MLNs) is a state-of-the-art SRL model that supports inference and querying over a probabilistic knowledge base (KB) defined by sets of entities, classes, relations, uncertain facts and uncertain first-order rules [4]. Table 1 shows an example probabilistic KB constructed from web extractions. The rule set $\mathcal{L}$ defines an MLN, allowing us to infer facts that are not explicitly stated in the original KB. These facts are uncertain, and their joint probability distribution is represented by a ground PGM (factor graph), as shown in Figure 2. The task of grounding and probabilistic inference is a key challenge we address in this paper. Based on the inference result, queries such as "Return all writers lived in Brooklyn" over this probabilistic KB would produce a list of person names ranked by the probabilities.

Entities $\mathcal{E}$	**Classes** $\mathcal{C}$	**Relations** $\mathcal{R}$	**Facts** Π
Ruth Gruber, New York City, Brooklyn	W (Writer) = {Ruth Gruber}, C (City) = {New York City}, P (Place) = {Brooklyn}	born in(W, P), born in(W, C), live in(W, P), live in(W, C), locate in(P, C)	0.96 born in(Ruth Gruber, New York City) 0.93 born in(Ruth Gruber, Brooklyn)

Rules $\mathcal{L}$
1.40 $\forall x \in W\ \forall y \in P$ (live in$(x, y) \leftarrow$ born in(x, y))
1.53 $\forall x \in W\ \forall y \in C$ (live in$(x, y) \leftarrow$ born in(x, y))
0.32 $\forall x \in P\ \forall y \in C\ \forall z \in W$ (locate in$(x, y) \leftarrow$ live in$(z, x) \wedge$ live in(z, y))
0.52 $\forall x \in P\ \forall y \in C\ \forall z \in W$ (locate in$(x, y) \leftarrow$ born in$(z, x) \wedge$ born in(z, y))
∞ $\forall x \in C\ \forall y \in C\ \forall z \in W$ (born in$(z, x) \wedge$ born in$(z, y) \rightarrow x = y$)

Table 1: A probabilistic graph data model representation of a sample probabilistic KB.

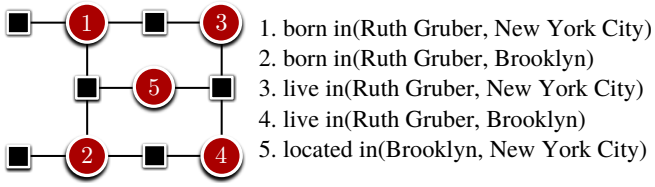


Figure 2: Factor graph representation of probabilistic KB in Table 1.

3 Model Representation and Model-Data Join

In a probabilistic knowledge base (KB), as the example in Table 1, queries need to be performed over the propositional form of the first-order PGM (MLN). The process of generating the propositional KB graph is called *grounding* in the SRL literature. The grounding process replaces variables in the first-order rules with constants, *materializing* it into a propositional PGM (factor graph), as shown in Figure 2, for further inference and querying.

The main challenge in KB graph materialization is efficiency and scalability due to the fast-growing sizes of current knowledge bases. For instance, the YAGO2s knowledge base [18] has 10 million entities and 120 million facts, and Freebase [19] has 45 million entities and 2.77 billion facts. To effectively manage these knowledge bases, recent works, PROBKB [4] and TUFFY [9], use bottom-up relational implementations to take the best advantage of query execution and optimization in a data-parallel processing system such as a parallel database. New techniques are also invented to further reduce the propositional KB graph based on deterministic semantic constraints [4].

3.1 Materialization via Model-Data Joins

As shown in [20], A naive materialization of the propositional KB graph enumerates all possible constants for every free variable in each first-order rule, which results in an exponential blow-up in the number of ground atoms and clauses. However, previous work [21, 22] have shown that this is both impractical and unnecessary in an extremely sparse relational domain, where only a very small fraction of the ground atoms are true and a vast majority of ground clauses are trivially satisfied given evidence and the closed world assumption (CWA). Based on these ideas, [9] implements a lazy closure grounding algorithm to generate a reduced grounded factor graph from an MLN program using a bottom-up grounding strategy.

While such a bottom-up approach greatly improves efficiency and scalability compared to top-down inference engines such as Alchemy [23], evaluation results show that the bottleneck lies in the number of first-order rules. This is because [9] represents each predicate as a table and translates each first-order rule into a recursive SQL query over the corresponding predicate tables. Given tens of thousands of rules over a KB, this implies tens of thousands of SQL queries in each iteration, incurring an unnecessarily heavy workload.

In our work on knowledge base expansion [4], we tackled this problem by exploiting the structure of first-order rules, partitioning them into tables, each representing a set of structurally equivalent rules. Instead of storing the MLNs in normal text files, the rules are stored as relational tables inside the database, and as a consequence, the grounding algorithm can be efficiently expressed as join queries among the facts and rules tables that apply inference rules *in batches*. Our experiments used the REVERB-SHERLOCK Wikipedia dataset [24, 25], which contains more than $400K$ facts, $270K$ entities and $30K$ rules. We categorized the first-order Horn clauses of lengths 2 and 3 and all functional dependency constraints into structurally equivalent rule tables [4]. During grounding, we ran one SQL query for each unique rule structure. The experiment results showed orders-of-magnitude speed-up compared to the state-of-the-art MLN inference engine. As a SQL-based algorithm, our approach allows easy integration of future improvements using query optimization and parallel databases like Greenplum.

3.2 Propositional KB Graph Reduction

Even with the lazy grounding techniques, the resulting propositional KB graph can still be prohibitively large, especially with large numbers of recursive first-order rules. However, due to the ambiguous entities and uncertain facts and rules, many of the inferred facts are erroneous. To make it worse, these erroneous facts propagate rapidly in the inference chain without proper constraints, as shown in Figure 3.

To improve the accuracy, we use probability thresholds to prune the rules with low credibility and a set of deterministic semantic constraints to detect erroneous facts and ambiguous entities. Before grounding starts, we eliminate rules with low probabilities from the rule tables. This avoids unsound rules applied repeatedly to multiple facts. Then, in each grounding iteration, we remove facts violating the deterministic semantic constraints to prevent them from further propagation.

Our experiments using REVERB-SHERLOCK and LEIBNIZ datasets [24, 25, 26] show that such probability and constraint based pruning produces a propositional KB graph with more than twice as many correct inferred facts. The accuracy of the inferred facts is improved to 60% from the 15% generated from the baseline. However,

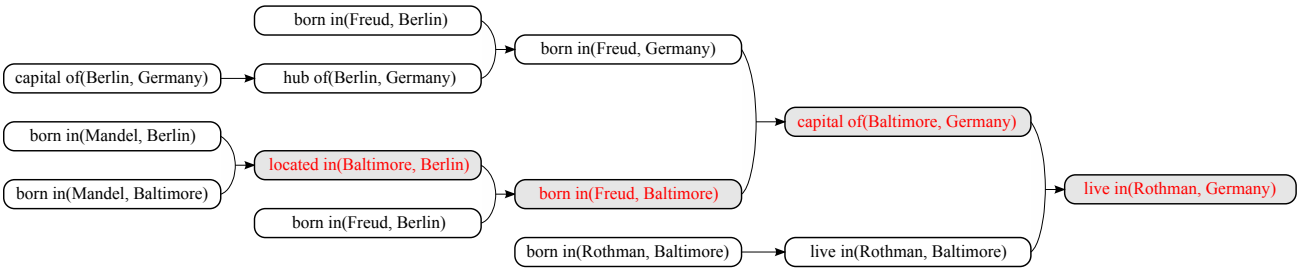


Figure 3: Error propagation: erroneous facts inferring other erroneous facts in the inference chain (shaded).

it is a nontrivial task to make further improvements since the extracted facts and rules are inherently noisy. Thus, instead of performing the pruning during inference, we propose, as a future work, to apply our constraining techniques to the rule learning phase to improve rule quality.

4 Efficient In-Database Statistical Inference

In many domains, structured data and unstructured text are both important assets for data analytics. For example, electronic medical record (EMR) systems use transactional databases to store both structured data such as lab results and monitor readings, and unstructured data such as notes from physicians. PGM-based methods such as linear-chain CRF, the example shown in Figure 1(a), are used for information extraction (IE) from text. With our goal to push PGM-based analytics close to the data, it is important to support basic statistical methods, such as feature extraction and inference for in-database text analytics tasks.

Basic text analytics tasks include part-of-speech (POS) tagging, named entity extraction (NER), and entity resolution (ER) [27]. Different statistical models and algorithms are implemented for each of these tasks with different runtime-accuracy tradeoffs. For example, an entity resolution task could be to find all mentions in a text corpus that refer to a real-world entity X. Such a task can be done efficiently by approximate string matching to find all mentions in text that approximately match the name of entity X. However, such a method is not as accurate as the state-of-the-art collective entity resolution algorithms based on statistical models, such as conditional random fields (CRFs).

4.1 Statistical Text Analytics in MADlib

Based on the MADlib [5] framework, we set out to implement statistical methods in SQL to support various text analytics tasks. We use CRFs as the basic statistical model to perform more advanced text analytics. Similar to Hidden Markov Models (HMM), linear-chain CRFs are a leading probabilistic model for solving many text analytics tasks, including POS and NER. To support sophisticated text analytics, we first implement key methods for text feature extraction including approximate string matching.

Text Feature Extraction: Text feature extraction is a first step in most statistical text analytics methods, and it can be an expensive operation. To achieve high quality, CRF models often assign hundreds of features to each token in the document. Examples of such features include: (1) dictionary features: *does this token exist in a provided dictionary?* (2) regex features: *does this token match a provided regular expression?* (3) edge features: *is the label of a token correlated with the label of a previous token?* (4) word features: *does this the token appear in the training data?* and (5) position features: *is this token the first or last in the token sequence?* The right combination of the features depends on the application. We implement a common set of text feature extractors with exactly one SQL query for the extraction of one type of features from text data [28]. For example, the following SQL query extract all the features that are matches of one of the regular expressions present in the `regextable` table:

```
SELECT start_pos, doc_id, 'R_' || r.name, ARRAY[-1,label]
FROM regextbl r, segmenttbl s
WHERE s.seg_text ~ r.pattern;
```

Approximate String Matching: A recurring primitive operation in text processing applications is the ability to match strings approximately. The technique we use is based on q-grams [29]. Specifically, Approximate string matching between two named entities text strings (e.g., George W. Bush vs. George Bush) is used as one of the features in a PGM such as conditional random field (CRF) to perform entity resolution (i.e., co-reference). We use the trigram module in PostgreSQL to create and index 3-grams over text [30]. Given a string "Tim Tebow" we create a 3-gram by using a sliding window of 3 characters over the text string. Using the 3-gram index, we create an approximate matching user-defined function (UDF) that takes a query string and returns all documents in the corpus that contain at least one approximate match.

4.2 In-Database Inference over PGMs

Once we have the features, the next step is to perform inference on the model. We implement two types of statistical inference within the database: the Viterbi algorithm to compute the MAP (maximum-a-priori) labels over a liear-chain PGM over sequence data, and Markov chain Monte Carlo (MCMC) to compute the marginal probability distribution over a general PGM.

Viterbi Inference: The Viterbi dynamic programming algorithm is a popular algorithm to find the top-k most likely labelings of a document for linear chain CRF models [27].

Like any dynamic programming algorithm, the Viterbi algorithm is recursive. We experiment with two different implementations. First, we implement it with a combination of recursive SQL and window aggregate functions. We discuss this implementation at length in an earlier work [15]. Our initial recursive SQL implementation only runs over PostgreSQL versions 8.4 and later; it does not run in Greenplum. Second, we implement a Python UDF that uses iterations to drive the recursion in the Viterbi algorithm. This iterative implementation runs on both PostgreSQL and Greenplum. In Greenplum, Viterbi runs in parallel over different subsets of the document on a multi-core machine. We also implement a CRF learning algorithm based on convex optimization in parallel using Python-UDF interface. Our evaluation uses CoNLL2000 dataset containing 8936 tagged sentences for learning, which achieves a 0.9715 accuracy consistent with the state-of-the-art POS models with 45 POS tags. To evaluate the inference performance, we extract 1.2 million sentences from the New York Times dataset [31]. Results show that the runtime is sublinear to and improves with an increase in the number of cores [28].

MCMC Inference: Markov chain Monte Carlo (MCMC) algorithms are classical sampling methods that are used to estimate probability distributions. We implement two MCMC methods in MADlib: Gibbs sampling and Metropolis-Hastings (MCMC-MH).

The MCMC algorithms involve iterative procedures where the current iteration depends on previous iterations. We use SQL window aggregates to carry "states" across iterations to perform the Markov-chain process. This window function based implementation runs on PostgreSQL version 8.4 and later. We discuss this implementation at length in our recent work [7]. We are currently working on integrating MCMC algorithms into Greenplum DBMS. We also plan to implement MCMC using Python UDFs and extend user-defined aggregates (UDAs) to support a new class of general iterative state transformation (GIST) operations and compare the performance between the two new in-database implementations of MCMC sampling based inference algorithms over PGMs.

In addition to the above techniques, there are a host of other features of both PostgreSQL and MADlib that are valuable for statistical inference in text analytics. Extension libraries for PostgreSQL and Greenplum provide text processing features such as inverted indexes, trigram indexes for approximate string matching, and array data types to model parameters. Existing modules in MADlib, such as Naive Bayes and Sparse/Dense Matrix

manipulations are building blocks to implement statistical text analytics methods. Leveraging this diverse set of tools and techniques that are already in the database allows us to build a sophisticated text analytics engine with statistical inference that has comparable performance to off-the-shelf implementations, but runs natively in a DBMS close to the data [15, 7, 28].

5 Optimizing Queries with Inference

While most of the current systems perform PGM-based data analytics offline in a batch mode, more applications require online data analytics based on PGM methods including inference. As an example, a query over EMR data can be: *return all patients who had a heart operation in the past 3 years and who lives in Berkeley city*. Processing of this query requires linear-chain CRF-based extraction over address text fields because a simple inverted index lookup would return patient records who live on *"Berkeley"* street as well as those who live in *"Berkeley"* city.

Another query example is over the probabilistic KB in Figure 2: *Q: return all writers who lived in Brooklyn*. This query requires inference over the probabilistic KB graph, however, only on a very small portion of the KB graph that is relevant to the query. This observation applies for queries over linear-chain PGM-based text extractions as well: the inference incurred from a query can be applied to a small fraction of PGMs to get the desired answer.

The intuition is to focus the inference computation on subparts of a grounded PGM that are most correlated with the query results. First query-driven inference algorithms are developed [15, 32], which adapt batch-oriented inference algorithms to prune computation given relational query (SQL) conditions.

5.1 Query-Proportional Sampling-Based Inference

In probabilistic KB literature, query-driven inference is done via a top-down theorem proving algorithm [33]. There are two issues with this state-of-the-art approach. First, top-down theorem proving is very expensive as shown in previous work [33, 9]. Incremental theorem proving bounded by time, on the other hand, may miss variables that are influential (i.e., strongly correlated) to the variables in result. Second, the resulting proof tree can be prohibitively large.

To solve the first problem, the materialized propositional KB graph from Section 3 can be used to quickly identify all the variables correlated to the result. To solve the second problem, we can use query-proportional inference techniques. In a *query-proportional sampling algorithm*, the sampling focuses on nodes having strongest influence on the query nodes. An intuitive example of such influential nodes are the query nodes' direct neighbors. In this section, we introduce a quantitative approach to measure such *influence*.

Given a query Q, it is reasonable to choose a sample distribution p that more frequently selects the query variables for sampling. Clearly, the query variable marginal depends on the remaining variables, so we must tradeoff sampling between query and non-query variables. Observing that not all non-query variables influence the query variables equally, we need to answer a key question: *How to pick a variable selection distribution p for a query Q to obtain the highest fidelity answer under a finite time budget?*

Previous work proposes to sample variables based on their *influence* on a query variable according to the graphical model defined by generalizing mutual information [32]. The mutual information measures dependence as a distance between the full joint distribution and its independent approximation. If variables x and y are independent, then this distance is zero and so is their mutual information. Such a query-proportional sampling algorithm based on the influence function not only increases the efficiency of sampling-based inference algorithms, but also computes best-effort approximate result given a time budget.

Our current work has shown promising results applying query-proportional sampling in a cross-document coreference application. We perform query-proportional sampling for coreference queries where only one or few

entities need to be resolved. The result shows that query-driven inference converges for low-selectivity queries in seconds to minutes compared to hours over the New York Times dataset [31].

5.2 Constrained Inference Optimizations

Apart from query-proportional sampling algorithms, another class of query optimization techniques are based on constrained inference. Existing inference algorithms can be optimized by pushing both *query-specific* and *model-specific* constraints to restrict the possible worlds sampled and returned by the inference algorithms over probabilistic graphical models.

Query-specific constraints include the selection, subgraph matching and join conditions. Query-specific constraints limit the space of possible results. Prior work have pushed selection conditions into a Viterbi inference algorithm for information extraction from text [6]. In that case, Viterbi stops early if the selection condition cannot be satisfied in the top-k results, leading to faster query answering [6]. The selection condition over text extraction can be `WHERE token = 'apple' and label = 'company'`. Such selection conditions over text would be translated into the positions of specific `apple` tokens with their doc ID's and label ID's.

Example 1: As an example of how we push a selection condition into a Viterbi matrix V in Figure 4, consider the condition: return text string d with the street num as the label of token #2 (counting from 0). Then in the second recursive step, only the partial segmentation in $V(1, streetnumber)$ satisfies the condition. In the third recursive step, because no partial segmentations in $V(2, y)$, $y \in Y$ come from cell $V(1, streetnumber)$ as shown in Figure 4, token #2 is the smallest pruning position. Thus, we stop the dynamic programming algorithm and conclude that d does not satisfy the condition.

pos	street num	street name	city	state	country
0	5	1	0	1	1
1	2	15	7	8	7
2	12	24	21	18	17
3	21	32	34	30	26
4	29	40	38	42	35
5	39	47	46	46	50

Figure 4: Viterbi matrix V for the linear-chain CRF model in Figure 1(a). First row are all the possible labels and first column are the token positions in the text string "2181 Shattuck North Berkeley CA USA" starting from 0. Each cell $V(i, y)$ stores a ranked list of *entries* $e = \{score, prev(label, idx)\}$ ordered by *score*. Each entry in $V(i, y)$ contains: (1) score of a top-k (partial) segmentation ending at position i with label y; and, (2) a pointer to the previous entry $prev$ on the path that led to top-k scores in $V(i, y)$.

Model-specific constraints refer to the deterministic or near deterministic correlations between variables in a PGM. Such constraints limit the space of possible worlds for sampling, which can be explored to reduce inference computation. Inference algorithms such as MCSAT have been proposed to effectively handle deterministic constraints in relational domain [34] for first-order PGMs such as MLN.

A final query optimization approach is a cost-based heuristic to choose among different PGM inference algorithms, either exact or approximate based on the query and the statistics of the grounded PGM model. Prior work has shown promising results of a hybrid inference algorithm for skip-chain CRF over text and probabilistic database queries over extraction results [7]. The evaluation shows that a simple cost-based optimizer can achieve significant speed-up by choosing the more efficient inference algorithms for different data, model and query combinations.

6 Conclusion

In this article, we discuss challenges and approaches in extending database systems for efficient and scalable in-database analytics based on probabilistic graphical models. First, we discuss our recent work in database representation of probabilistic KBs and an efficient SQL-based grounding algorithm for first-order PGMs. Second, to support in-database statistical methods and inference, we discuss the implementation of statistical text analysis methods and PGM inference algorithms as part of the MADlib project, a library with native implementations of statistical methods in database. Finally, for efficient query processing with both relational and statistical inference operations, we discuss query-driven inference and constraint inference techniques as well as a cost-based optimizer to choose among different inference algorithms based on data, model and query.

Supporting PGM methods in database and online queries with statistical inference is a new and active research area. PGM methods are used for advanced modeling and analysis over very different types of data, including text data, relational data and graph data. We expect much more progress made and techniques developed in this area in the near future.

References

[1] D. Borthakur, "The hadoop distributed file system: Architecture and design," *Hadoop Project Website*, vol. 11, p. 21, 2007.

[2] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *NSDI*, 2012.

[3] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "Graphlab: A new framework for parallel machine learning," in *UAI*, 2010.

[4] Y. Chen and D. Z. Wang, "Knowledge expansion over probabilistic knowledge bases," in *SIGMOD Conference*, pp. 649–660, 2014.

[5] J. Hellerstein, C. Re, F. Schoppmann, D. Wang, E. Fratkin, A. Gorajek, K. Ng, C. Welton, X. Feng, K. Li, and A. Kumar, "The madlib analytics library or mad skills, the sql," *To appear in VLDB*, 2012.

[6] D. Z. Wang, M. J. Franklin, M. Garofalakis, and J. M. Hellerstein, "Querying probabilistic information extraction," *VLDB*, 2010.

[7] D. Z. Wang, M. J. Franklin, M. Garofalakis, J. M. Hellerstein, and M. L. Wick, "Hybrid in-database inference for declarative information extraction," in *SIGMOD*, 2011.

[8] J. Cohen, B. Dolan, M. Dunlap, J. M. Hellerstein, and C. Welton, "Mad skills: new analysis practices for big data," *Proc. VLDB Endow.*, vol. 2, pp. 1481–1492, 2009.

[9] F. Niu, C. Ré, A. Doan, and J. Shavlik, "Tuffy: scaling up statistical inference in markov logic networks using an rdbms," *VLDB*, 2011.

[10] P. Sen and A. Deshpande, "Representing and querying correlated tuples in probabilistic databases," in *ICDE*, pp. 596–605, 2007.

[11] D. Z. Wang, E. Michelakis, M. Garofalakis, and J. M. Hellerstein, "Bayesstore: managing large, uncertain data repositories with probabilistic graphical models," *In Proceedings of VLDB Endowment*, vol. 1, no. 1, pp. 340–351, 2008.

[12] M. Wick, A. McCallum, and G. Miklau, "Scalable probabilistic databases with factor graphs and mcmc," *In Proceedings of VLDB Endowment*, 2010.

[13] R. Jampani, F. Xu, M. Wu, L. L. Perez, C. Jermaine, and P. J. Haas, "Mcdb: a monte carlo approach to managing uncertain data," in *Proceedings of the 2008 ACM SIGMOD*, pp. 687–700, 2008.

[14] H. C. Bravo and R. Ramakrishnan, "Optimizing mpf queries: Decision support and probabilistic inference," in *SIGMOD*, 2007.

[15] D. Wang, E. Michelakis, M. Franklin, M. Garofalakis, and J. Hellerstein, "Probabilistic Declarative Information Extraction," in *ICDE*, 2010.

[16] D. Koller and N. Friedman, *Probabilistic graphical models: principles and techniques*. The MIT Press, 2009.

[17] C. Sutton and A. McCallum, "Introduction to conditional random fields for relational learning," in *Introduction to Statistical Relational Learning*, 2008.

[18] J. Biega, E. Kuzey, and F. M. Suchanek, "Inside yago2s: a transparent information extraction architecture," in *Proceedings of the 22nd international conference on World Wide Web companion*, International World Wide Web Conferences Steering Committee, 2013.

[19] K. Bollacker, C. Evans, P. Paritosh, T. Sturge, and J. Taylor, "Freebase: a collaboratively created graph database for structuring human knowledge," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, ACM, 2008.

[20] M. Richardson and P. Domingos, "Markov logic networks," *Machine learning*, vol. 62, no. 1, pp. 107–136, 2006.

[21] J. Shavlik and S. Natarajan, "Speeding up inference in markov logic networks by preprocessing to reduce the size of the resulting grounded network," in *Proceedings of the 21st International Jont Conference on Artifical Intelligence*, IJCAI'09, pp. 1951–1956, Morgan Kaufmann Publishers Inc., 2009.

[22] P. Singla and P. Domingos, "Memory-efficient inference in relational domains," in *Proceedings of the 21st National Conference on Artificial Intelligence - Volume 1*, AAAI'06, pp. 488–493, AAAI Press, 2006.

[23] S. Kok, M. Sumner, M. Richardson, P. Singla, H. Poon, D. Lowd, J. Wang, and P. Domingos, "The alchemy system for statistical relational AI," tech. rep., 2009.

[24] "ReVerb dataset, http://reverb.cs.washington.edu/,"

[25] S. Schoenmackers, O. Etzioni, D. Weld, and J. Davis, "Learning first-order horn clauses from web text," in *EMNLP*, 2010.

[26] T. Lin and O. Etzioni, "Identifying functional relations in web," *In Proceeding of Conference on Empirical Methods in Natural Language Processing*, 2010.

[27] C. D. Manning and H. Schütze, *Foundations of statistical natural language processing*. MIT Press, 1999.

[28] K. Li, C. Grant, D. Z. Wang, S. Khatri, and G. Chitouras, "Gptext: Greenplum parallel statistical text analysis framework," in *Proceedings of the Second Workshop on Data Analytics in the Cloud*, pp. 31–35, ACM, 2013.

[29] L. Gravano, P. G. Ipeirotis, H. V. Jagadish, N. Koudas, S. Muthukrishnan, L. Pietarinen, and D. Srivastava, "Using q-grams in a dbms for approximate string processing," 2001.

[30] C. E. Grant, J. dan Gumbs, K. Li, D. Z. Wang, and G. Chitouras, "Madden: query-driven statistical text analytics," in *CIKM*, pp. 2740–2742, 2012.

[31] "New York Times dataset, https://catalog.ldc.upenn.edu/ldc2008t19,"

[32] M. L. Wick and A. McCallum, "Query-aware mcmc," in *NIPS*, 2011.

[33] S. Schoenmackers, O. Etzioni, and D. Weld, "Scaling textual inference to the web," in *EMNLP*, 2008.

[34] H. Poon, P. Domingos, and M. Sumner, "A general method for reducing the complexity of relational inference and its application to mcmc," in *Proceedings of the 23rd National Conference on Artificial Intelligence - Volume 2*, AAAI'08, pp. 1075–1080, AAAI Press, 2008.

SystemML's Optimizer: Plan Generation for Large-Scale Machine Learning Programs

Matthias Boehm, Douglas R. Burdick, Alexandre V. Evfimievski, Berthold Reinwald,
Frederick R. Reiss, Prithviraj Sen, Shirish Tatikonda, and Yuanyuan Tian

IBM Research – Almaden, San Jose, CA, USA

Abstract

SystemML enables declarative, large-scale machine learning (ML) via a high-level language with R-like syntax. Data scientists use this language to express their ML algorithms with full flexibility but without the need to hand-tune distributed runtime execution plans and system configurations. These ML programs are dynamically compiled and optimized based on data and cluster characteristics using rule- and cost-based optimization techniques. The compiler automatically generates hybrid runtime execution plans ranging from in-memory, single node execution to distributed MapReduce (MR) computation and data access. This paper describes the SystemML optimizer, its compilation chain, and selected optimization phases for generating efficient execution plans.

1 Introduction

Large-scale machine learning has become critical to the success of many business applications such as customer experience analysis, log analysis, social data analysis, churn analysis, cyber security, and many others. Both, ever increasing data sizes and analysis complexity naturally lead to large-scale, data and task parallel ML.

SystemML [2, 5, 11] aims at declarative, large-scale ML via a compiler-based approach. ML programs such as regression, classification, or clustering are expressed in a high-level language with R-like syntax called DML (Declarative Machine learning Language). SystemML compiles these programs, applies rewrite rules and cost-based optimizations according to data and cluster characteristics, and generates runtime execution plans. Runtime plans may include in-memory, single node computations and parallel computations on MapReduce clusters. The high-level language significantly increases the productivity of data scientists compared to low-level programming in MapReduce because it provides full flexibility and data independence as well as efficiency and scalability via automatic optimization. The compiler-based approach of SystemML differs from existing work on large-scale ML libraries like Mahout [10], MADlib [6] or MLlib [9], which mostly provide fixed algorithms and runtime plans and often expose physical data representations. However, there is also recent work like Cumulon [7] or future plans of Mahout [4] that follow SystemML towards declarative, large-scale ML.

Our DML language allows to express a rich set of ML algorithms ranging from descriptive statistics, classification, clustering, regression, to matrix factorization. To demonstrate the complexity of compiling arbitrary ML programs, we discuss two examples of regression and classification algorithms, and highlight their core computations. Optimizing these computations is key to algorithm performance and scalability.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

Example 1 (Linear Regression (Normal Equations) using $\mathbf{X}^\top\mathbf{X}$): Linear regression predicts a dependent continuous variable $\mathbf{y}$ from multiple continuous feature variables $\mathbf{X}$ by finding coefficients $\beta_0, \beta_1, \ldots, \beta_m$ using a regularization constant $\lambda > 0$. This formulation has the following matrix linear equation form: $\mathbf{A}\beta = \mathbf{X}^\top\mathbf{y}$, where $\mathbf{A} = \mathbf{X}^\top\mathbf{X} + \text{diag}(\lambda)$. In SystemML's DML syntax, this computation is expressed as:

```
1:  A = t(X) %*% X + diag(lambda);
2:  b = t(X) %*% y;
3:  beta = solve(A, b);
```

where `solve` is a built-in function for solving a linear system of equations that takes a small matrix $\mathbf{A}$ and a vector $\mathbf{b}$, and returns the coefficient vector β. Regarding data sets with many observations, i.e. rows, computing $\mathbf{X}^\top\mathbf{X}$ and $\mathbf{X}^\top\mathbf{y}$ are by far the most expensive operations. First, for $\mathbf{X}^\top\mathbf{X}$, a dedicated implementation method could be applied which exploits the unary input characteristic avoiding data shuffling, and leveraging local partial aggregation, as well as the result symmetry by doing only half the computation. Second, for $\mathbf{X}^\top\mathbf{y}$, assuming that vector $\mathbf{y}$ fits in memory, we can distribute $\mathbf{y}$ via Hadoop distributed cache and again avoid data shuffling and leverage local partial aggregation. We can also rewrite $\mathbf{X}^\top\mathbf{y}$ to $(\mathbf{y}^\top\mathbf{X})^\top$ in order to prevent the transpose of $\mathbf{X}$ altogether. In this example, all operations can be packed into a single MR job that reads $\mathbf{X}$ only once.

Example 2 (Logistic Regression via Trust Region Method): Our second running example is logistic regression that learns a binary classifier $h : \Re^m \rightarrow \{\pm 1\}$ where h is expressed in terms of a linear function $h(x) = 1$ if $\beta^\top x \geq 0$ and $h(x) = -1$ otherwise. The goal is then to learn the model parameters β. Given a fully labeled training set $\{(x_i, y_i)\}$, where x_i denotes the i^{th} example and $y_i \in \{\pm 1\}$ its label, one can learn β with regularization constant λ by minimizing the following objective function: $\lambda/2\beta^\top\beta + \sum_i \log\left[1 + \exp(y_i\beta^\top x_i)\right]$. There are numerous ways to perform the above optimization. The trust-region Newton method works well in high-dimensional feature spaces [8] and can be expressed in DML as follows (simplified):

```
 1:  while( sum(g^2) > exit_g2 & i < max_i ) {
 2:     sb = zeros_nf;  r = g;  r2 = sum(r^2);  exit_r2 = 0.01 * r2;
 3:     d = -r;  tb_reached = FALSE; j = 0;
 4:     while( r2 > exit_r2 & (! tb_reached) & j < max_j ) {
 5:        Hd = lambda * d + t(X) %*% diag(v) %*% X %*% d;    # core computation
 6:        a = r2 / sum(d * Hd);
 7:        [a,tb_reached] = ensure_tb(a,sum(d^2),2*sum(sb*d),sum(sb^2)-delta^2);
 8:        sb = sb + a * d;  r = r + a * Hd;  r2_new = sum(r^2);
 9:        d = - r + (r2_new / r2) * d;  r2 = r2_new; j = j + 1;  }
10:     p = 1.0 / (1.0 + exp(-y * (X %*% (b + sb))));
11:     so = -sum(log (p)) + 0.5 * lambda * sum((b + sb)^2) - o; i = i + 1;
12:     delta = update_tr(delta,sqrt(sum(sb^2)),so,sum(sb*g),0.5*sum(sb*(r+g)));
13:     if( so < 0 ) {
14:        b = b + sb;  o = o + so;  v = p * (1 - p);
15:        g = - t(X) %*% ((1 - p) * y) + lambda * b;  }
16:  }
```

In the above ML program, we learn a logistic regression classifier using two nested loops. This template of two nested loops is similar to many implementations of unconstrained optimization problems. The inner `while` loop uses an iterative solver to compute the optimal update to model parameters β. Note that the subroutines `ensure_tb` and `update_tr` ensure trust bounds and update the trust region but are non-essential from a computational perspective. The core computation of the inner loop is $\mathbf{X}^\top(diag(\mathbf{v})\mathbf{X}\mathbf{d})$ (line 5). Analyzing the data flow, we can exploit important characteristics. First, since $\mathbf{X}\mathbf{d}$ produces a column vector, we can rewrite the expression to $\mathbf{X}^\top(\mathbf{v} \cdot \mathbf{X}\mathbf{d})$, eliminating the `diag` operation and replacing the subsequent matrix multiplication with a cell-wise vector multiplication of $\mathbf{v} \cdot (\mathbf{X}\mathbf{d})$. Second, since a block in $\mathbf{X}$ is equivalent to the related block in $\mathbf{X}^\top$, we can—if $\mathbf{v}$ and $\mathbf{d}$ fit in memory—invoke a dedicated operator that reads $\mathbf{v}$ and $\mathbf{d}$ through Hadoop distributed cache and does a single pass over $\mathbf{X}$ to compute the entire matrix multiplication chain.

In general, SystemML parses and compiles these ML programs to hybrid runtime execution plans. Runtime operations range from in-memory, single node operations to large-scale cluster operations via MapReduce jobs. Hybrid execution plans enable us to scale up and down as required by input data and program characteristics.

System Architecture: The overall SystemML architecture has different layers. The *language* contains a rich set of statistical functions, linear algebra operations, control structures, user-defined and external functions, recursion, and ML-specific operations. The parser produces directed acyclic graphs (DAGs) of *high-level operators* (HOPs) per block of statements as defined by control structures. HOPs such as matrix multiplication, unary and binary operations, or reorg operations operate on intermediates of matrices and scalars. Various optimizations are applied on HOP DAGs, including operator ordering and selection. Operator selection is significant as—similar to relational query optimization—the runtime supports for expensive operations such as matrix multiplication, several alternative physical operators, which are chosen depending on data and cluster characteristics. HOP DAGs are transformed to *low-level operator* (LOP) DAGs. Low-level operators such as grouping, aggregate, transform, or binary operations operate on runtime-specific intermediates such as key-value pairs in MapReduce. Given assigned LOP execution types, the LOPs of a DAG are piggybacked into a workflow of MR jobs in order to minimize data scans, for operator pipelining, and latency reduction. LOPs have equivalent *runtime* implementations, i.e., runtime instructions which are either executed in-memory of the single node control program (CP) or on MapReduce (MR). MR instructions are executed via few generic MR job types.

Discussion SystemML Design: The overall goal of SystemML is to allow declarative ML for a wide variety of use cases, where ML algorithms can be implemented independent of input data and cluster characteristics. Four major requirements influenced the design of SystemML. First, the need for *full flexibility* in specifying new ML algorithms and customizing existing algorithms led to a domain-specific language for ML. A library interface of ML algorithms could not achieve this level of flexibility. Second, the choice of operations in our DML language was directly influenced by the goal of *data independence*. Semantic operations such as linear algebra operations make us independent from the underlying physical representation such as dense/sparse representations, row/column major layout, or blocking configurations. In contrast to general purpose parallel programming languages, the DML approach simplifies optimization because the operation semantics are preserved and with that they are easier to reason about. The approach also simplifies the runtime as alternative physical operator implementations may be provided. Third, the requirement of running the same ML algorithms on very small to very large data sets combined with the need for *efficiency and scalability* demands automatic optimization and hybrid runtime plans of in-memory single node operations, and large-scale cluster computations. Fourth, our current view on declarative ML is focused on *specified algorithm semantics* and execution plan generation for performance and scalability only. We do not optimize for accuracy because this would contradict the language flexibility and would make it harder to understand, debug, and control the algorithm behavior.

2 Compilation Chain Overview

SystemML uses a well defined compilation chain in order to generate an executable runtime program (execution plan) for a given DML script. Figure 1 shows an overview of this chain and associates the individual compilation and optimization phases to SystemML's general architecture, where the example DML expression on the right refers back to Example 2 (Logistic Regression, line 10). Our overall optimization objective is to minimize the script execution time under hard memory constraints given a certain cluster configuration.

Language-Level: First, we start the compilation process by *parsing* the given DML script into a hierarchical representation of statement blocks and statements, where statement blocks are defined by the program structure in terms of control flow constructs like branches, loops, or calls to user defined functions. This parsing step is responsible for lexical and syntactic analysis but also for aspects like basic order of operations (e.g., multiply/plus). In detail, we use a parser generator to create this parser based on our specific DML grammar. Second, we do a classic *live variable analysis* [1] in terms of a data flow analysis over statement blocks. We

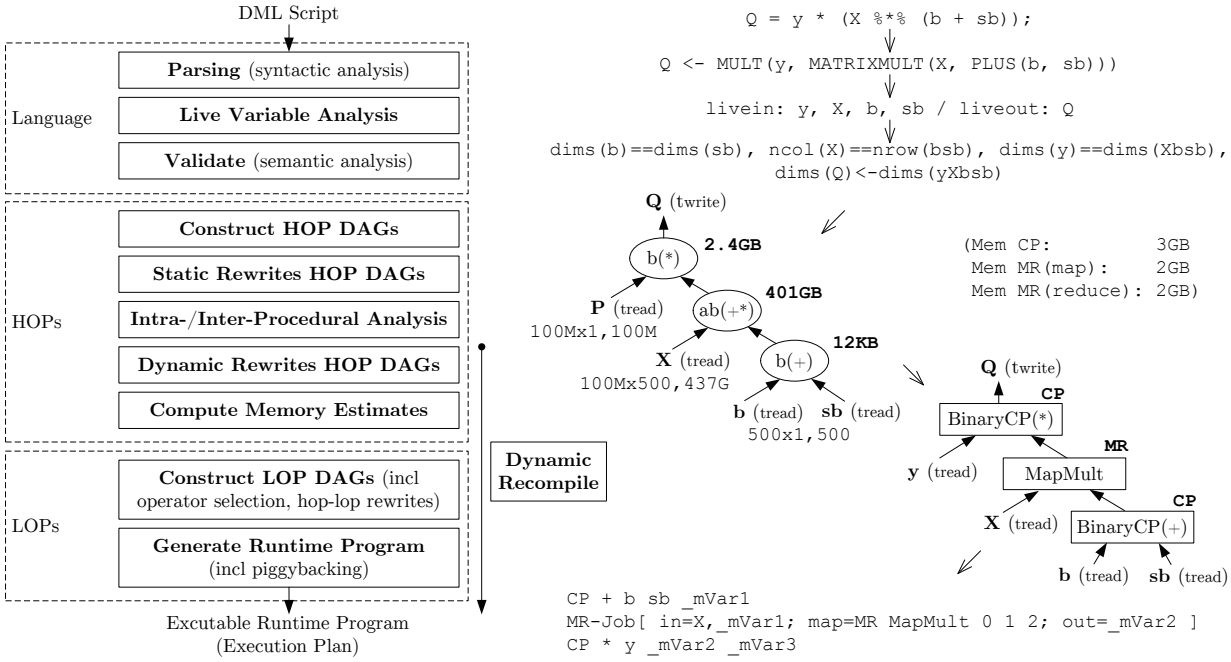


Figure 1: Overview of SystemML's Compilation Chain with Examples.

obtain, for example, livein and liveout variable sets in a forward and backward pass over the entire program. The third compilation step at language level is *validate*, where we perform a semantic analysis of the program and its expressions. In contrast to the previous two steps, this validation is already inherently ML-domain specific. Examples for expression validation are compatibility checks of dimensions and checks for mandatory parameters of built-in functions. This validate step is done in one pass over the entire program, where we recursively validate all statement blocks, statements and expressions. During this pass, we also do an initial constant and size (data characteristic) propagation, with awareness of size updates in conditional branches or loops.

HOP-Level (Subsections 3.1 and 3.2): For each basic block of statements—i.e., predicates and last-level statement blocks—we then create DAGs of high-level operators. Nodes represent (logical) operations and their outputs (scalar/matrix with specific value type), while edges represent data dependencies. First, we recursively *construct* a single operator tree for all statements and expressions of this statement block. Second, after all DAGs have been constructed, we apply *static HOP DAG rewrites*. These rewrites comprise all size-independent rewrites, i.e., program and HOP DAG transformations that are always required or always (assumed to be) beneficial. Examples are mandatory format conversions, common subexpression elimination (CSE), constant folding, algebraic simplifications, and branch removal. Third, we subsequently perform an *intra-/inter-procedural analysis*, where we propagate size information over the—now rewritten—program structure and into functions. Note that we only propagate sizes into functions if they are called with consistent sizes of arguments. Fourth, we apply *dynamic HOP DAG rewrites*, which cover all size-dependent HOP rewrites. These rewrites include (a) simplifications that are always beneficial but only apply under certain size conditions, and (b) cost-based rewrites that require sizes for cost estimation. Examples are matrix multiplication chain optimization and dynamic algebraic simplifications. Fifth, based on propagated sizes, we compute worst-case *memory estimates* for all HOPs. These estimates reflect the memory consumption of this operation with regard to in-memory execution in CP. Figure 1 (middle right) shows the HOP DAG of the example expression annotated with size information and memory estimates. This HOP DAG includes two binary HOPs b(+) and b(*) for cell-wise vector addition and multiplication, respectively, as well as an aggregate binary HOP ab(+*) for matrix multiplication.

LOP-Level (Subsections 3.3 and 3.4): With the rewritten HOP DAGs and computed memory estimates, all preconditions for runtime plan generation are available. We generate runtime plans via DAGs of low-level operators, where nodes represent (physical) operations and edges represent again data dependencies. Individual LOPs are runtime-backend-specific, but a DAG can contain mixed LOP types of integrated runtimes like CP and MR. First, we *construct LOP DAGs* for all HOP DAGs. The compiler uses a recursive traversal (with memoization) to perform this transformation. The recursion, which starts at the DAG root nodes, involves multiple steps per operator: We start with LOP operator selection, where we decide between CP and MR under hard constraints w.r.t. memory estimates and budget as well as on specific physical operators. If sizes are unknown, we fall back to robust MR operators but mark the current HOP for recompilation. Subsequently, we apply specific HOP-LOP rewrites that inherently depend on which physical operator we have chosen. Finally, the individual LOPs are constructed and configured. Second, once the LOP DAG is complete, we *generate the runtime program*. This phase of compilation includes the generation of executable program blocks per statement block and instructions for each LOP DAG. During instruction generation, we also perform piggybacking of multiple MR LOPs into composite MR jobs. Piggybacking is done via a greedy bin packing algorithm that satisfies additional constraints such as map/reduce execution location and memory constraints. Finally, we obtain an executable runtime program, where CP and MR instructions are wrappers of matrix operations which work on entire matrices or matrix blocks respectively. Figure 1 (bottom right) shows the LOP DAG of the example expression. For both binary HOPs, we create `BinaryCP` LOPs as the sizes are small given a CP memory budget of $3\,GB$. For the aggregate binary HOP, however, we create a `MapMult` LOP since the operation memory of $401\,GB$ is too large for the CP memory budget but the vector input ($4\,KB$) fits into the map task budget of $2\,GB$.

Runtime-Level (Section 4): During runtime, we *dynamically recompile* HOP DAGs as required. Recompilation comprises all compilation steps from dynamic rewrites down to runtime instructions as shown in Figure 1 (Dynamic Recompile). This dynamic recompilation is our robust fallback strategy whenever we are not able to propagate size or sparsity during initial compilation, and it is very important due to high latency of unnecessary MR jobs and high impact of selecting the right physical operators on large data. Note that for average scripts, language-level compilation requires about $200\,ms$ per script and HOP-level compilation takes about $10\,ms$ per DAG; while recompilation (including LOP-level) is by design $< 1\,ms$ per DAG because it is part of the critical path. Finally, there are additional runtime optimizers like the `parfor` optimizer [2] that includes task-parallelism-specific rewrites and works with a global optimization scope over the entire loop body.

3 Selected Optimization Phases

We now describe essential optimization phases of SystemML's compilation chain in detail. This description includes (1) static and dynamic HOP DAG rewrites, (2) size propagation and memory estimates, (3) operator selection and HOP-LOP rewrites, as well as (4) runtime plan generation via piggybacking.

3.1 Static and Dynamic Rewrites

Fundamentally, we distinguish two rewrite categories: static (size independent) and dynamic (size dependent), which are both applied at the HOP level to enable reuse across runtime backends. Examples for static rewrites comprise common subexpression elimination, constant folding, static algebraic simplifications, and branch removal. In addition, examples for dynamic rewrites are matrix multiplication chain optimization and dynamic algebraic simplifications. In the following, we discuss these essential rewrites in detail.

Common Subexpression Elimination (CSE): During HOPs construction, we created an initial operator tree. CSE then identifies and removes redundant operations. There are two sources of redundancy: (1) from the original specification (see for example `(1-p)` in Example 2, line 14/15), and (2) resulting from other rewrites.

Our simple yet efficient approach consists of two steps. First, we collect and replace all leaf nodes—i.e. variable reads and literals—via a unique name-operator map. Second, we recursively remove common subexpressions bottom-up starting at these leaf nodes. For each node, we compare and merge parent nodes if they have the same inputs and equivalent configurations. This step is done in an operator-aware fashion in order to prevent invalid eliminations of non-deterministic operators like `rand` with unspecified seed.

Constant Folding: Constant folding mostly serves as a preparation step for size propagation and more sophisticated rewrites. We fold sub-DAGs of operations over literals into a single literal by—similar to existing work [1]—compiling and executing runtime instructions. This elegant approach ensures consistency, which is especially important with regard to type promotion as well as NaN and overflow handling.

Static Algebraic Simplifications: Static algebraic simplifications are size-independent rewrites, since they are assumed to be always beneficial. As shown in Equation 5, this category of rewrites includes the removal of unnecessary operations, the transformation from binary to unary operations because they are easier to parallelize, and specific simplifications that either reduce the number of intermediates or change the asymptotic complexity:

$$\begin{aligned}
\text{Remove Unnecessary Operations:} \quad & \mathbf{X}^{\top\top}, \quad \mathbf{X}/1, \quad \mathbf{X}\cdot 1, \quad \mathbf{X}-0 \rightarrow \mathbf{X}, \quad matrix(1,\ldots))/\mathbf{X} \rightarrow 1/\mathbf{X} \\
& rand(\ldots, \text{min}=-1, \text{max}=1)\cdot 7 \rightarrow rand(\ldots, \text{min}=-7, \text{max}=7) \\
\text{Binary to Unary:} \quad & \mathbf{X}+\mathbf{X} \rightarrow 2\cdot\mathbf{X}, \quad \mathbf{X}\cdot\mathbf{X} \rightarrow \mathbf{X}^2, \quad \mathbf{X}-\mathbf{X}\cdot\mathbf{Y} \rightarrow \mathbf{X}\cdot(1-\mathbf{Y}) \\
\text{Simplify Diag Aggregates:} \quad & sum(diag(\mathbf{X})) \rightarrow trace(\mathbf{X}), \quad trace(\mathbf{XY}) \rightarrow sum(\mathbf{X}\cdot\mathbf{Y}^{\top})
\end{aligned} \tag{5}$$

We apply those rewrites after CSE because—especially for binary to unary rewrites—this order greatly simplifies the rewrite framework since we only need to probe local inputs for identity. Finally, we do a second CSE pass.

Remove Branches: As a preparation for size propagation, we remove branches with constant conditions. This rewrite happens after constant folding, which folds arbitrary constant predicate expressions into a boolean constant. In case of true conditions, we replace the entire if-else block with the if branch body, while for false conditions, we either use the else branch body or remove again the entire block. Removing, for instance, branches that append a column of 1s to $\mathbf{X}$ for models with intercept allows us to propagate unconditional sizes.

Matrix Multiplication Chain Optimization: The associative order of a chain of matrix multiplications strongly affects sizes of intermediates and computational effort. Consider for example $\mathbf{X}^{\top}(diag(\mathbf{v})\mathbf{Xd})$ from Example 2, line 5: computing $\mathbf{X}^{\top}diag(\mathbf{v})$ first would produce an intermediate in the size of $\mathbf{X}$, while performing these operations in the right order produces only vector intermediates. Since this optimization problem exhibits the properties of *optimal substructure* and *overlapping subproblems*, we use a classic dynamic programming algorithm. This bottom-up algorithm computes the cost-optimal parentheses for subchains and composes them to the overall chain, where we currently assume independence with regard to the sparsity of intermediates.

Dynamic Algebraic Simplification Rewrites: Dynamic algebraic simplifications are size dependent rewrites, i.e., rewrites which use sizes in validity constraints or for cost comparison. This category includes the removal of unnecessary indexing operations, simplifications for structural aggregations, the removal of operations with empty output, and simplifications of operations involving vectors.

$$\begin{aligned}
\text{Remove Unnecessary Indexing:} \quad & \mathbf{X}=\mathbf{Y}[\ldots], \quad \mathbf{X}[\ldots]=\mathbf{Y} \rightarrow \mathbf{X}=\mathbf{Y}, \text{iff dims}(\mathbf{X})=\text{dims}(\mathbf{Y}) \\
& \mathbf{X}=\mathbf{Y}[\ldots] \rightarrow \mathbf{X}=matrix(0,\ldots), \text{iff nnz}(\mathbf{Y})=0 \\
\text{Simplify Aggregates:} \quad & colSums(\mathbf{X}), \quad rowSums(\mathbf{X}) \rightarrow sum(\mathbf{X}), \text{iff ncol/nrow}(\mathbf{X})=1 \\
\text{(sum, min, max, mean, prod)} \quad & colSums(\mathbf{X}), \quad rowSums(\mathbf{X}) \rightarrow \mathbf{X}, \text{iff nrow/ncol}(\mathbf{X})=1 \\
& sum(\mathbf{X}) \rightarrow 0, \quad colSums(\mathbf{X}) \rightarrow matrix(0,\ldots), \text{iff nnz}(\mathbf{X})=0 \\
\text{Remove Empty Operators:} \quad & \mathbf{X}+\mathbf{Y}, \quad \mathbf{X}-\mathbf{Y} \rightarrow \mathbf{X}, \text{iff nnz}(\mathbf{Y})=0 \\
& \mathbf{XY}, \quad \mathbf{X}\cdot\mathbf{Y}, \quad round(\mathbf{X}), \quad \mathbf{X}^{\top} \rightarrow matrix(0,\ldots), \text{iff nnz}(\mathbf{X}|\mathbf{Y})=0 \\
\text{Simplify Matrix Multiply:} \quad & diag(\mathbf{X})\mathbf{y} \rightarrow \mathbf{X}\cdot\mathbf{y}, \text{iff ncol}(\mathbf{y})=1, \text{otherwise } (\mathbf{X}matrix(1,...))\cdot\mathbf{y} \\
& diag(\mathbf{XY}) \rightarrow rowSums(\mathbf{X}\cdot t(\mathbf{Y})), \text{iff ncol}(\mathbf{y})=1 \\
& (-\mathbf{X}^{\top})\mathbf{y} \rightarrow -(\mathbf{X}^{\top}\mathbf{y}), \text{iff size}(\mathbf{X}^{\top}\mathbf{y}) < \text{size}(\mathbf{X})
\end{aligned} \tag{6}$$

If sizes are known during initial compilation, those rewrites are applied statically (in-place of the original DAG). Otherwise those rewrites take affect during dynamic recompilation with potential for a very aggressive rewrite scope. For instance, in Example 2, each outer iteration re-initializes **sb** to an empty matrix. Hence, in each first inner iteration, we can eliminate `2*sum(sb*d)` and `sum(sb^2)` (line 7) via dynamic rewrites.

3.2 Size Propagation and Memory Estimates

Size propagation and memory estimates are crucial for declarative ML with both in-memory and large-scale computation. Any memory estimates inherently need to reflect the underlying runtime. In our single node runtime, a multi-level buffer pool controls in-memory objects, and runtime instructions pin their inputs/outputs in memory. This pinning prevents serialization in operations like matrix multiply that access data multiple times or out-of-order. However, to prevent out-of-memory situations, we need to guarantee that memory estimates never underestimate. We now describe size propagation over ML programs and worst-case memory estimates.

Intra-/Inter-Procedural Analysis (IPA): Important size information for memory estimates are matrix dimensions and sparsity. Input sizes of `reads` are given by meta data that is mandatory for sparse matrix formats. Both *validate* and *IPA* then propagate these sizes over the entire ML program. Here, we use IPA as an example. IPA aims at propagating sizes into and across DML-bodied functions. In general, we use a candidate-based algorithm. First, we determine candidate functions by collecting all function calls and their input parameters. Second, we prune all functions that are called with potentially different dimension sizes of input matrices. Third, for all remaining positive candidates, we determine if we can also safely propagate sparsity into that function. Fourth, we do a full intra-/inter-procedural analysis over the entire program. In detail, we iterate over the hierarchy of statement blocks. For each statement block, we push input variable sizes into DAG leaf nodes, recursively propagate them bottom-up and in an operator-aware manner through the DAG, and finally extract the result variables of this DAG. IPA also pays special care to conditional control structures. For if conditions, we only propagate the size of a variable if both branches lead to the same size. For loops, we determine if sizes change in the loop body; and if they do, we re-propagate unknown sizes into the loop body. Whenever, we hit a function call that is marked as an IPA candidate, we recursively propagate sizes into that function and after that continue intra-procedure analysis with the resulting function output sizes.

Memory Estimates: We then recursively compute memory estimates—-with memoization—for entire HOP DAGs. These estimates reflect single-threaded, in-memory computations. Starting at leaf nodes (reads and literals), we compute the output memory estimate according to the HOP output sizes via a precise model of our sparse and dense matrices [2]. If the sparsity is unknown, we use dense as the worst case since we only switch to sparse if smaller. We then compute output estimates for all non-leaf HOPs, after which we compute operation memory estimates using the sum of all child outputs, intermediates, and output estimate. For example, the operation memory estimate of a binary operator for cell-wise matrix addition would include the output memory of both inputs and its own output. During this process, we also propagate worst-case sparsity and dimensions according to the operator semantics. For example, for a $[m \times k, s_1] \times [k \times n, s_2]$ matrix multiply, we use a worst case sparsity estimate of $s_3 = \min(1, s_1 k) \cdot \min(1, s_2 k)$ although the average case estimate would be $s_3 = 1 - (1 - s_1 s_2)^k$. Additionally, we backtrack size constraints. For example, for row-wise left indexing into a matrix R, the input matrix size is known to be of size $[1 \times ncol(R)]$. Finally, these memory estimates are used for all in-memory matrices, matrix blocks, and operations over all of them.

3.3 Operator Selection and Ordering

Operator selection determines the best execution strategy for every HOP in a DAG. The selected plan is a sub-DAG of LOPs with certain physical properties but we retain the HOP DAG for later recompilation. Operator selection includes (1) selecting the execution type, (2) selecting execution-type-specific physical operators, and

(3) local rewrites that are dependent on the physical operator or computed memory estimates.

LOP Selection (Execution Type and Physical Operators): The translation from HOPs to LOPs is done in a local manner on a per-HOP basis. For a given HOP, the selection of LOPs is determined by four factors: (1) memory budget for CP, (2) memory estimate of the HOP, (3) data characteristics such as dimensions and sparsity, and (4) cluster characteristics such as degree of parallelism and memory budget of mappers/reducers. Based on the HOP memory estimate, we first decide on the execution type, i.e. single-node CP or MR. Currently, we follow the heuristic that in-memory operations require less time than their MR counterparts and hence, select CP whenever the memory estimate is smaller than the budget. This heuristic can be relaxed by incorporating cost models of execution time for generated runtime plans. Second, we determine the exact runtime strategy. For example, we currently support five physical operators for MR matrix multiplication: `TSMM` (for $\mathbf{X}^\top\mathbf{X}$), `MapMult` (for $\mathbf{Xv}$, if $\mathbf{v}$ fits into the mapper memory budget), `MapMultChain` (for entire chains of $\mathbf{X}^\top(\mathbf{w} \cdot \mathbf{Xv})$ or $\mathbf{X}^\top(\mathbf{Xv})$ if $\mathbf{w}$ and $\mathbf{v}$ fit into the mapper memory budget), as well as `CPMM` and `RMM` (as previously described in [5]). The decision on physical operators is governed by a combination of cost functions and heuristics, where it is important to take the resulting degree of parallelism into account. For common use cases like Examples 1 and 2, we typically compile `TSMM`, `MapMult`, or `MapMultChain`, which nicely fit the MR model in terms of core computation in mappers, local aggregation via combiners, and very small final aggregation.

HOP-LOP Rewrites: After the decision on physical operators, we apply HOP-LOP rewrites that depend on the chosen LOPs. Examples are decisions on MR aggregation, empty block materialization, and input partitioning but also rewrites like transpose re-ordering. Let us use $\mathbf{X}^\top\mathbf{y}$ from Example 1 (line 2), where $\mathbf{X}$ is $[10^8 \times 500]$ and assume we decided for MR MapMult. First, since multiple row blocks (10^5 for a blocksize of $[10^3 \times 10^3]$) contribute to the result, we need to add another LOP for local aggregation. Second, since $\mathbf{y}$ requires already $800\,MB$ and needs to be read sideways through distributed cache, we introduce a LOP for data partitioning. If $\mathbf{y}$ fits in the CP memory budget this is done in CP; otherwise in MR. Third, we reorder $\mathbf{X}^\top\mathbf{y}$ to $(\mathbf{y}^\top\mathbf{X})^\top$ if $\mathbf{y}$ and the result are smaller than $\mathbf{X}$ and both introduced transpose operations fit in the CP memory budget. The partition LOP is accordingly configured as column block partitioning. Fourth, because the output of MapMult is consumed only by the CP transpose, we configure the MapMult to not output empty blocks. In summary, even for a single HOP, there are many different execution plans, depending on data and cluster characteristics.

3.4 Piggybacking

We generate the runtime program via piggybacking. The input to this compilation step is the previously constructed LOP DAG and the output is an ordered list of runtime instructions. This list of instructions includes in-memory CP instructions and MR-Job instructions. Piggybacking packs multiple MR instructions into shared MR jobs for scan sharing (data-dependent) and latency reduction (independent). Currently, our piggybacking optimization objective is to minimize the number of MR jobs, but other objectives like the amount of read/written data are possible. In the following, we describe the core algorithm (an extension of the previous description [5]).

Piggybacking MR LOPs into MR Jobs: Conceptually, the problem of packing instructions into a minimal number of MR jobs is a bin packing problem with multiple constraints. Example constraints are (1) job-type requirements, (2) the execution location of operations (map, reduce, map and/or reduce, CP), (3) if the operation changes keys (breaks block alignment), (4) if the operation groups keys (aligns blocks), and (5) the memory consumption. Our algorithm has two steps. First, we do a topological sort of the LOP DAG. Second, we do a greedy grouping of operations via an extended *next fit* bin packing heuristic. The algorithm operates in multiple *rounds*. In each round, it maintains multiple open MR jobs, one for each job type. It subsequently assigns LOPs to these open MR jobs as long as all constraints are satisfied. Once all possible LOPs are assigned in the current round, the MR jobs are closed and corresponding MR-Job instructions are generated. We repeat this process until all LOPs are assigned. This algorithm has linear best-case and quadratic worst-case complexity in the number of LOPs and thus allows for efficient runtime plan generation.

4 Dynamic Recompilation

The described compilation techniques work very well if we can infer intermediate sizes and sparsity, which is true for simple iterative algorithms where the training data $\mathbf{X}$ is accessed read-only and thus all important operations are known. For other types of programs, the sizes or sparsity of intermediates may be unknown during initial compilation. We use dynamic recompilation as our robust fallback strategy for these cases.

Example scenarios are scripts with functions, sampling, data dependent operations, and changing dimensions or sparsity. First, if functions are called with arguments of different sizes, the plan of the function itself, the function outputs, and subsequent operations are unknown. The same limitation also applies to external UDFs. Second, sampling—as used for fold creation in cross validation or pre-sampling large data—leads to unknowns for the entire training algorithm as well. For instance, consider a simple sampling via permutation matrices:

```
1:   s  = ppred( rand(rows=nrow(X), cols=1, min=0, max=1), 0.01, "<=" );
2:   Xs = removeEmpty( diag(s), margin="rows" ) %*% X;  #take an ~1% sample of X
```

where `ppred` creates a sample indicator vector $\mathbf{s}$. This allows us to sample $\mathbf{X}$ with a single matrix multiplication but due to randomization leads to an unknown number of rows of the output matrix $\mathbf{Xs}$. Third, data-dependent operations like `table` (computes contingency tables), `aggregate` (computes grouped aggregates) or `removeEmpty` (removes empty rows or columns) pose challenges for size inference because even the dimensions become unknown or are heavily overestimated. Fourth, changing dimensions or sparsity also lead to unknowns because we need to conservatively consider the worst case.

Dynamic recompilation inherently shares the same goals and challenges as adaptive query processing [3] in DBMSs. Specifically, dynamic recompilation is always a trade-off between the potential overhead of breaking pipelines and benefit of re-optimization. However, we can exploit specific characteristics of large-scale ML programs. First, conditional control flow requires us to split DAGs anyway into statement blocks with materialized intermediates. Second, the re-optimization potential is huge. Operations on small data execute in milliseconds in memory while we would pay 10s latency for MR jobs. In addition, picking the right physical operators is crucial on large data as well. These characteristics allow a robust and simple, yet very effective recompiler that is used by default. In what follows, we describe (1) compile time decisions, and (2) recompilation during runtime.

4.1 Optimizer Recompilation Decisions

Dynamic recompilation works at the granularity of HOP DAGs in order to exploit natural block boundaries and simplify compilation to unconditional blocks. The optimizer makes two important decisions, namely (1) on the recompilation granularity by splitting HOP DAGs, and (2) on whether or not to mark HOP DAGs for recompilation. This control allows us to trade the flexibility of an interpreter with the efficiency of a compiler.

Split HOP DAGs for Recompilation: Our major goal is to prevent unknowns but keep DAGs as large as possible to exploit all piggybacking opportunities. Hence, we follow a very conservative approach of splitting HOP DAGs for recompilation: by default, we do not split any DAGs. Exceptions are persistent reads with unknown sizes (without provided metadata) and specific data-dependent operations like `table` where our worst-case estimates largely overestimate. This DAG split is done either during initial parsing or as a static HOP rewrite rule if dependent on additional information. In detail, we collect these operators and their inputs, replace them with transient reads and put them into a new statement block which is inserted just before the original one.

Mark HOP DAGs for Recompilation: Marking DAGs for recompilation is part of operator selection during LOP DAG construction. Our simple strategy is to mark a HOP for recompilation whenever we select conservative physical operators (MR, or CP out-of-core) due to unknown sizes or sparsity. This means that most DAGs that include at least one MR job are marked for recompilation. This policy is advantageous in any case because the large latency of an MR job or disk I/O makes recompilation overhead negligible. For large data this policy allows the compiler to change physical operators, and for small data we might be able to compile pure in-memory

instructions. We do not mark CP operations for recompilation—even if the sparsity is unknown—because it is unknown if the overhead of recompilation can be amortized by the benefits of dynamic rewrites. Subsequently, we aggregate recompilation flags of all HOPs to the DAG level: if at least one HOP requires recompilation, the entire DAG requires recompilation and we materialize this information.

Finally, note that the recompiler is used for consistency in many related optimization phases. For example, we currently exploit the recompiler for size propagation in intra-/inter procedure analysis, as a preparation step in runtime optimizers like `parfor`, and for cost estimation of runtime plans in more sophisticated optimizers.

4.2 Dynamic Recompilation at Runtime

Dynamic recompilation during runtime is done at specific recompilation hooks. The most important hooks exist before execution of last-level program block instructions and predicate instructions. Just before executing the initially compiled instructions, we access the statement block associated with a program block; if it is marked for recompilation, we compile the HOP DAG by leveraging the current symbol table with metadata and statistics of all live variables. In detail, dynamic recompilation of a single HOP DAG comprises the following steps:

- **Deep Copy DAG:** First, we create a deep copy of the given HOP DAG in order to apply non-reversible dynamic rewrites but keep the original DAG for later recompilations.
- **Update DAG Statistics:** Second, we update DAG statistics by updating leaf node statistics according to the current symbol table. Subsequently, we recursively propagate those statistics bottom-up through the DAG, where each HOP updates its output statistics according to its input statistics and semantics. For operators with size expressions (e.g., `rand`, `seq`, `reshape`, range indexing) we also evaluate simple expressions similar to constant folding but with regard to the current values of the symbol table.
- **Dynamic Rewrites:** Third, we apply the dynamic rewrites described earlier. Having the symbol table with exact statistics allows us to apply very aggressive rewrites that would be invalid in general. Examples are first iterations of certain algorithms with zero initialized model coefficients.
- **Recompute Memory Estimates:** Fourth, we recompute the memory estimates according to the updated statistics and rewritten DAG. With this unconditional scope of a single DAG, we typically obtain very good estimates because we already split DAGs where we would heavily overestimate.
- **Generate Runtime Instructions:** Fifth, we construct LOPs (including operator selection and HOP-LOP rewrites) and generate runtime instructions (including piggybacking). All these substeps are heavily dependent on the memory estimates and hence benefit from updated statistics.

5 Conclusions

We showed an overview of SystemML's compilation chain, selected optimization phases, and dynamic recompilation. Declarative large-scale ML with a high-level domain-specific language aims at both (1) flexibility in specifying new large-scale machine learning algorithms as well as (2) efficiency and scalability via size-aware automatic optimization. Inherently, the optimizer is key because it allows users to write their algorithms once and deploy them in many settings with different or even unknown input characteristics. Our major directions for future work comprise (1) optimizer support for next generation runtime platforms like YARN and Spark, (2) optimizer support for hardware accelerators like many core co-processors or GPUs, and (3) global data flow optimization. We strongly believe that both growing analysis complexity and growing data sizes will further increase the importance of automatic optimization. Hence, we would like to encourage the database community to contribute to this emerging domain of declarative machine learning.

References

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers – Principles, Techniques, & Tools*. Addison-Wesley, 2007.

[2] M. Boehm, S. Tatikonda, B. Reinwald, P. Sen, Y. Tian, D. Burdick, and S. Vaithyanathan. Hybrid Parallelization Strategies for Large-Scale Machine Learning in SystemML. *PVLDB*, 7(7):553–564, 2014.

[3] A. Deshpande, Z. G. Ives, and V. Raman. Adaptive Query Processing. *Foundations and Trends in Databases*, 1(1):1–140, 2007.

[4] Dmitriy Lyubimov. *Mahout Scala Bindings and Mahout Spark Bindings for Linear Algebra Subroutines*. The Apache Software Foundation, 2014. `http://mahout.apache.org/users/ sparkbindings/home.html`.

[5] A. Ghoting, R. Krishnamurthy, E. P. D. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian, and S. Vaithyanathan. SystemML: Declarative Machine Learning on MapReduce. In *ICDE*, 2011.

[6] J. M. Hellerstein, C. Ré, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li, and A. Kumar. The MADlib Analytics Library or MAD Skills, the SQL. *PVLDB*, 5(12):1700–1711, 2012.

[7] B. Huang, S. Babu, and J. Yang. Cumulon: Optimizing Statistical Data Analysis in the Cloud. In *SIGMOD*, 2013.

[8] C.-J. Lin, R. C. Weng, and S. S. Keerthi. Trust Region Newton Method for Logistic Regression. *Journal of Machine Learning Research*, 9:627–650, 2008.

[9] E. R. Sparks, A. Talwalkar, V. Smith, J. Kottalam, X. Pan, J. E. Gonzalez, M. J. Franklin, M. I. Jordan, and T. Kraska. MLI: An API for Distributed Machine Learning. In *ICDM*, 2013.

[10] The Apache Software Foundation. *Mahout*.

[11] Y. Tian, S. Tatikonda, and B. Reinwald. Scalable and Numerically Stable Descriptive Statistics in SystemML. In *ICDE*, 2012.

Tupleware: Distributed Machine Learning on Small Clusters

Andrew Crotty, Alex Galakatos, Tim Kraska
Brown University
{crottyan, agg, kraskat}@cs.brown.edu

Abstract

There is a fundamental discrepancy between the targeted and actual users of current analytics frameworks. Most systems are designed for the challenges of the Googles and Facebooks of the world—petabytes of data distributed across large cloud deployments consisting of thousands of cheap commodity machines. Yet, the vast majority of users operate clusters ranging from a few to a few dozen nodes, analyze relatively small datasets of up to several terabytes in size, and perform primarily compute-intensive operations. Targeting these users fundamentally changes the way we should build analytics systems.

This paper describes our vision for the design of Tupleware, a new system specifically aimed at performing complex analytics (e.g., distributed machine learning) on small clusters. Tupleware's architecture brings together ideas from the database and compiler communities to create a powerful end-to-end solution for data analysis. Our preliminary results show orders of magnitude performance improvement over alternative systems.

1 Introduction

The growing prevalence of big data across all industries and sciences is causing a profound shift in the nature and scope of analytics. Increasingly complex computations such as machine learning (ML) are quickly becoming the norm. However, current analytics frameworks (e.g., Hadoop [1], Spark [36]) are designed to meet the needs of giant Internet companies; that is, they are built to process petabytes of data in cloud deployments consisting of thousands of cheap commodity machines. Yet non-tech companies like banks and retailers—or even the typical data scientist—seldom operate clusters of that size, instead preferring smaller clusters with more reliable hardware. In fact, recent industry surveys reported that the median Hadoop cluster was fewer than 10 nodes, and over 65% of users operate clusters smaller than 50 nodes [20, 28].

Furthermore, the vast majority of users typically analyze relatively small datasets. For instance, the average Cloudera customer rarely works with datasets larger than a few terabytes in size [15], and commonly analyzed behavioral data peaks at around 1TB [11]. Even companies as large as Facebook, Microsoft, and Yahoo! frequently perform ML tasks on datasets smaller than 100GB [32]. Rather, as users strive to extract more value than ever from their data, computational complexity becomes the true problem.

Targeting more complex workloads on smaller clusters fundamentally changes the way we should design analytics tools. Current frameworks disregard single-node performance and instead focus on the major challenges of large cloud deployments, in which data I/O is the primary bottleneck and failures are common [17]. In fact, as

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

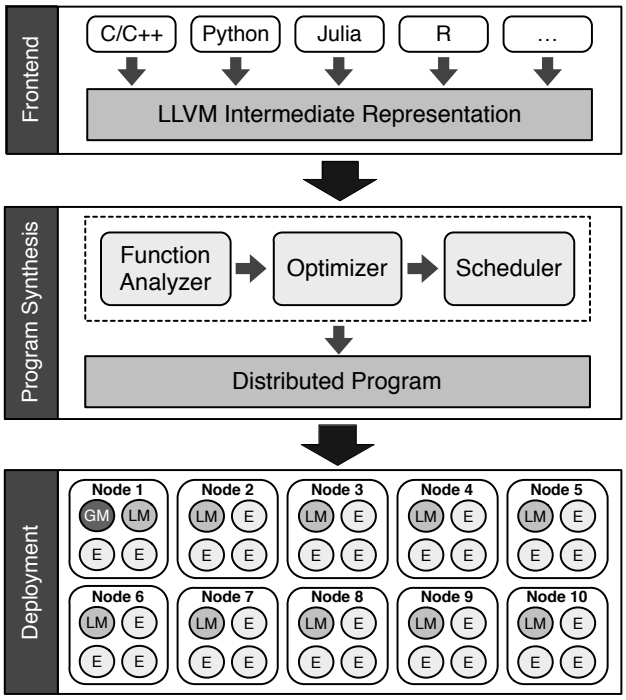


Figure 1: An overview of Tupleware's architecture, which depicts the three distinct yet interrelated components of the system: (1) frontend, (2) program synthesis, and (3) deployment.

our experiments show, it is surprising to see how inefficiently these frameworks utilize the available computing resources.

In this paper, we describe our vision for the design of Tupleware, a high-performance distributed system built specifically for performing complex analytics on small clusters. The key idea behind Tupleware is to integrate high-level DBMS optimizations with low-level compiler optimizations in order to best take advantage of the underlying hardware. Tupleware compiles workflows comprised of *user-defined functions* (UDFs) directly into a distributed program. More importantly, workflow compilation allows the system to apply optimizations on a case-by-case basis that consider properties about the data, computations, and hardware together.

2 System Overview

Tupleware is a distributed, in-memory analytics platform that targets complex computations, such as distributed ML. The system architecture is shown in Figure 1 and is comprised of three distinct parts.

Frontend: Tupleware allows users to define ML workflows directly inside a host language by supplying UDFs to API operators such as map and reduce. Our new algebra, based on the strong foundation of functional programming with monads, seeks a middle ground between flexibility and optimizability while also addressing the unique needs of ML algorithms. Furthermore, by leveraging the LLVM [25] compiler framework, Tupleware's frontend is language-agnostic, and users can choose from a wide variety of programming languages (visualized as the top boxes in Figure 1) with little associated overhead. We describe Tupleware's algebra and API in Section 3.

Program Synthesis: When the user submits a workflow to Tupleware, the *Function Analyzer* first examines each UDF to gather statistics for predicting execution behavior. The *Optimizer* then converts the workflow into

a self-contained distributed program and applies low-level optimizations that specifically target the underlying hardware using the previously gathered UDF statistics. Finally, the *Scheduler* plans how best to deploy the distributed program on the cluster given the available resources. The program synthesis process is explained in Section 4.

Deployment: After compiling the workflow, the distributed program is automatically deployed on the cluster, depicted in Figure 1 as ten nodes (shown as boxes) each with four hyperthreads (circles inside the boxes). Tupleware utilizes a multitiered deployment setup, assigning specialized tasks to dedicated threads, and also takes unique approaches to memory management, load balancing, and recovery. We discuss all of these deployment aspects further in Section 5.

3 Frontend

Ideally, developers want the ability to concisely express ML workflows in their language of choice without having to consider low-level optimizations or the intricacies of distributed execution. In this section, we describe how Tupleware addresses these points.

3.1 Motivation

Designing the right abstraction for ML tasks that balances usability and functionality is a tricky endeavor. We believe that a good programming model for ML has three basic requirements.

Expressive & Optimizable: MapReduce [17] is a popular programming model for parallel data processing that consists of two primary operators: a *map* that applies a function to every key-value pair, and a *reduce* that aggregates values grouped by key. Yet, many have criticized MapReduce, in particular for rejecting the advantages of high-level languages like SQL [4]. However, SQL is unwieldy for expressing ML workflows, resulting in convoluted queries that are difficult to understand and maintain. Other recent frameworks (e.g., Spark, Stratosphere [21], DryadLINQ [35]) have started to bridge the gap between expressiveness and optimizability by allowing users to easily compose arbitrary workflows of UDFs.

Iterations: Many ML algorithms are most naturally expressed iteratively, but neither MapReduce nor SQL effectively supports iteration [26, 16]. The most straightforward solution to this problem is to handle iterations via an external driver program. Both Spark and DryadLINQ take this approach, but the downside is that a completely independent job must be submitted for each iteration, making cross-iteration optimization difficult. In contrast, a number of iterative extensions to MapReduce have been proposed (e.g., Stratosphere, HaLoop [10], Twister [18]), but these approaches either lack low-level optimization potential or do not scale well.

Shared State: No existing framework incorporates an elegant and efficient solution for the key ingredient of ML algorithms: shared state. Many attempts to support distributed shared state within a MapReduce-style framework impose substantial restrictions on how and when programs can interact with global variables. For instance, the Iterative Map-Reduce-Update [9] model supplies traditional map and reduce functions with read-only copies of global state values that are recalculated during the update phase after each iteration. However, this paradigm is designed for iterative refinement algorithms and could be difficult to adapt to tasks that cannot be modeled as convex optimization problems (e.g., neural networks, maximum likelihood Gaussian mixtures). Furthermore, Iterative Map-Reduce-Update precludes ML algorithms that explore different synchronization patterns (e.g., Hogwild! [31]). Spark is another framework that supports shared state via objects called accumulators, which can be used only for simple count or sum aggregations on a single key, and their values cannot be read from within the workflow. Spark also provides broadcast variables that allow machines to cache large read-only values to avoid redistributing them for each task, but these values can never be updated. Therefore, broadcast variables cannot be used to represent ML models, which change frequently.

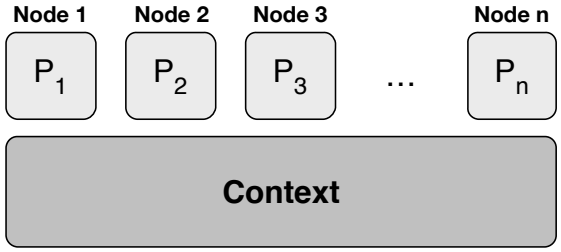


Figure 2: A visualization of a TupleSet's logical data model. Partitions $P_1, ..., P_n$ of the relation R are spread across n nodes, whereas the Context is logically shared across all nodes.

Type	Operator	λ-Function
Relational	selection$(T)(\lambda)$	$t \rightarrow b$
	projection$(T)(\lambda)$	$t \rightarrow t'$
	cartesian(T_1, T_2)	-
	θ-join$(T_1, T_2)(\lambda)$	$(t_1, t_2) \rightarrow b$
	union(T_1, T_2)	-
	difference(T_1, T_2)	-
Apply	map$(T)(\lambda)$	$(t, C) \rightarrow t'$
	flatmap$(T)(\lambda)$	$(t, C) \rightarrow \{t'\}$
	filter$(T)(\lambda)$	$(t, C) \rightarrow b$
Aggregate	reduce$(T)(\lambda)(\kappa?)$	$(t, C) \rightarrow (\Delta_\kappa, C')$
Control	load()	-
	evaluate(T)	-
	save(T)	-
	loop$(T)(\lambda)$	$C \rightarrow b$
	update$(T)(\lambda)$	$C \rightarrow C'$

Figure 3: A subset of TupleSet operators, showing their types and λ-function contracts.

3.2 Programming Model

Based on these requirements, we envision a new programming model for distributed ML that: (1) strikes a middle ground between the expressiveness of MapReduce and optimizability of SQL; (2) natively handles iterative workflows in order to optimize transparently across iterations; and (3) provides flexible shared state primitives with configurable synchronization patterns that can be directly accessed from within the workflow. Tupleware introduces an algebra based on the foundation of functional programming with monads to address all three of these points. We define this algebra on a data structure called a *TupleSet* comprised of a data relation and its associated *Context*, which is a dictionary of key-value pairs that stores the shared state. *Operators* define the different ways in which users can transform a TupleSet, returning a new TupleSet as output.

Tupleware's programming model allows for automatic and efficient parallel data processing. As shown in Figure 2, each node in the cluster processes a disjoint subset of the data. However, unlike other paradigms, Tupleware's API incorporates the notion of global state that is logically shared across all nodes.

3.3 Operator Types

We divide operators into four distinct types. Figure 3 shows the most common Tupleware operators, most of which take as input one or more TupleSets T, as well as the signatures of their associated λ-functions. The λ-functions are supplied by the user and specify the workflow's computation.

Relational: Relational operators include all of the traditional SQL transformations. For example, the user can perform a *selection* by passing a predicate UDF to the corresponding operator. As given in Figure 3, the expected UDF signature has the form: $t \rightarrow b$ where $t \in R$ and b is a Boolean value; that is, the user composes a (potentially compound) predicate using the set of operations $\{=, \neq, >, \geq, <, \leq\}$ that returns `true` if a given tuple t of the incoming relation R should be selected for the output relation R' and `false` otherwise. Note that relational operators interact only with the relation R of the TupleSet and cannot modify shared state variables maintained by the Context C. Relational operators therefore introduce no dependencies, so we can perform the standard query optimization techniques (e.g., predicate pushdown, join reordering). Note, though, that operators such as *θ-join* and *union* merge Context variables but do not change their values, performing SQL-style disambiguation of conflicting keys.

Apply: Apply operators invoke the supplied UDF on every tuple in the relation. Tupleware's API provides three apply operators: *map*, *flatmap*, and *filter*. The map operator requires a UDF that specifically produces a *1-to-1* mapping (i.e., the UDF takes one input tuple and must return exactly one output tuple). The flatmap operator takes a UDF that produces a *1-to-N* mapping but is more difficult to optimize. The filter operator takes a UDF that produces a *1-to-(0:1)* mapping and is less restrictive than the relational selection operator, permitting arbitrary predicate logic. By distinguishing among these different types of apply operators, our programming model provides the system with additional information about the workflow, thereby allowing for greater optimization.

Aggregate: Aggregate operators perform an aggregation UDF on the relation. Similar to Spark, Tupleware's *reduce* operator expects a commutative and associative λ-function. These semantics allow for the efficient parallelization of computations like sum and count, which return an output relation R' consisting of one or more aggregated values. Users can optionally specify a key function κ that defines the group-by semantics for the aggregation. If no key function is provided, then the computation is a single-key reduce (i.e., all tuples have the same key). Additionally, reduce λ-functions can modify Context variables in different ways, which we describe further in Section 3.4.

Control: As their names suggest, the *load*, *evaluate*, and *save* operators actually load data into memory, execute a declared workflow, and persist the results to the specified location, respectively, returning a handle to the result as a new TupleSet that can then be used in a subsequent workflow. Notice, though, that this programming model can efficiently cache and reuse results across several computations. In order to support iterative workflows, which are common to ML algorithms, Tupleware also incorporates a *loop* operator. The loop operator models a tail recursive execution of the workflow while the supplied loop invariant holds, and the UDF has access to the Context for maintaining information such as iteration counters or convergence criteria. Finally, Tupleware's algebra provides an *update* operator that executes logically in a single thread and allows for direct modification of Context variables.

3.4 Context

Shared state is an essential component of distributed ML algorithms, which frequently involve iterative refinement of a global model. Tupleware expresses shared state using monads, which are an elegant way to handle side effects in a functional language. Monads impose a happened-before relation between operators; that is, an operator O that modifies Context variables referenced by another operator O' must be fully evaluated prior to evaluating O'.

Each type of operator has different rules that govern interaction with the Context. As previously mentioned, relational operators cannot access Context variables. Apply operators have read-only access to Context variables, allowing them to be fully parallelized. On the other hand, the update operator can directly modify Context variables because it executes logically in a single thread.

For aggregate operators, we define three different Context variable update patterns that can express a broad range of algorithms.

Parallel: Parallel updates must be commutative and associative. Conceptually, these updates are not directly applied, but rather added to an update set. After the operation completes, the deltas stored in the update sets are aggregated first locally and then globally, possibly using an aggregation tree to improve performance.

Synchronous: Synchronous updates require that each concurrent worker obtain an exclusive lock before modifying a Context variable. This pattern ensures that each worker always sees a consistent view of the shared state and that no updates are lost.

Asynchronous: Asynchronous updates allow workers to read from and write to a Context variable without first acquiring a lock. This pattern makes no guarantees about when changes to a Context variable will become visible to other concurrent workers and updates may be lost. Many algorithms can benefit from relaxing synchronization guarantees. For example, Hogwild! shows that asynchronous model updates can dramatically improve the performance of stochastic gradient descent. While asynchronous updates will occur nondeterministically

```
ATTR = 2                                  #2 attributes (x,y)
CENT = 3                                  #3 centroids
ITER = 20                                 #20 iterations

def kmeans(c):
 ts = TupleSet('data.csv', c)             #load file 'data.csv'
 ts = ts.map(distance)                    #get distance to each centroid
        .map(minimum)                     #find nearest centroid
        .reduce(reassign)                 #reassign to nearest centroid
        .update(recompute)                #recompute new centroids
        .loop(iterate)                    #perform 20 iterations
        .evaluate()                       #trigger computation
 return ts.context()['k']                 #return new centroids

def distance(t1, t2, c):
 t2.copy(t1, ATTR)                        #copy t1 attributes to t2
 for i in range(CENT):                    #for each centroid:
  t2[ATTR+i] = sqrt(sum(map(lambda        # compute and store distance
      m,n:(n-m)**2,c['k'][i],t1))

def minimum(t1, t2):
 t2.copy(t1, ATTR)                        #copy t1 attributes to t2
 m,n = min(m,n for n,m                    #find index of min distance
   in enumerate(t[:CENT]))
 t2[ATTR] = n                             #assign to nearest centroid

def reassign(t1, c):
 assign = t1[ATTR]                        #get centroid assignment
 for i in range(ATTR):                    #for each attribute:
  c['sum'][assign][i] += t1[i]            # compute sum for assign
 c['ct'][assign] += 1                     #increment count for assign

def recompute(c):
 for i in range(CENT):                    #for each centroid:
  for j in range(ATTR):                   # for each attribute:
   c['k'][i][j] =                         #  calculate average
      c['sum'][i][j]/c['ct'][i]

def iterate(c):
 c['iter'] += 1                           #increment iteration count
 return c['iter'] < ITER                  #check iteration count
```

Figure 4: A Tupleware implementation of k-means in Python.

during the execution of an individual operator, the final result of that operator should always be deterministic given valid inputs.

3.5 Language Integration

As mentioned previously, Tupleware allows users to compose workflows and accompanying UDFs in any language with an LLVM compiler. Presently, C/C++, Python, Julia, R, and many other languages have LLVM backends.

The system exposes functionality in a given host language via a TupleSet wrapper that implements the Tupleware operator API (see Figure 3). As long as the user adheres to the UDF contracts specified by the API, Tupleware guarantees correct parallel execution. A TupleSet's Context also has a wrapper that provides special accessor and mutator primitives (e.g., get, set). With the increasing popularity of LLVM, adding new languages is as simple as writing a wrapper to implement Tupleware's API.

3.6 Example

Figure 4 shows a Python implementation of the k-means clustering algorithm using Tupleware's API. K-means is an iterative ML algorithm that classifies each input data item into one of k clusters. In the example, the driver function `kmeans` defines the workflow using the five specified UDFs, where $t1$ is an input tuple, $t2$ is an output tuple, and c is the Context. Note that unlike other approaches, Tupleware can store the cluster centroids as Context variables.

Function	Type	Vectorizable	Compute Time		Load Time
			Predicted	Actual	
`distance`	map	yes	30	28	3.75
`minimum`	map	yes	36	38	7.5
`reassign`	reduce	no	16	24	5.62
`recompute`	update	no	30	26	0

Table 2: Function statistics for the k-means algorithm gathered by the Function Analyzer.

4 Program Synthesis

Once a user has submitted a workflow, the system (1) examines and records statistics about each UDF, (2) generates an abstract execution plan, and (3) translates the abstract plan into a distributed program. We refer to this entire process as *program synthesis*. In this section, we outline the different components that allow Tupleware to synthesize highly efficient distributed programs.

4.1 Function Analyzer

Systems that treat UDFs as black boxes have difficulty making informed decisions about how best to execute a given workflow. By leveraging the LLVM framework, Tupleware can look inside UDFs to gather statistics useful for optimizing workflows during the code generation process. The Function Analyzer examines the LLVM intermediate representation of each UDF to determine vectorizability, computation cycle estimates, and memory bandwidth predictions. As an example, Table 2 shows the UDF statistics for the k-means algorithm from Section 3.6.

Vectorizability: Vectorizable UDFs can use *single instruction multiple data* (SIMD) registers to achieve data level parallelism. For instance, a 256-bit SIMD register on an Intel E5 processor can hold 8×32-bit floating-point values, offering a potential $8\times$ speedup. In the k-means example, only the `distance` and `minimum` UDFs are vectorizable, as shown in Table 2.

Compute Time: One metric for UDF complexity is the number of CPU cycles spent on computation. CPI measurements [3] provide cycles per instruction estimates for the given hardware. Adding together these estimates yields a rough projection for total UDF compute time, but runtime factors (e.g., instruction pipelining, out-of-order execution) can make these values difficult to predict accurately. However, Table 2 shows that these predictions typically differ from the actual measured compute times by only a few cycles.

Load Time: Load time refers to the number of cycles necessary to fetch UDF operands from memory. If the memory controller can fetch operands for a particular UDF faster than the CPU can process them, then the UDF is referred to as *compute-bound*; conversely, if the memory controller cannot provide operands fast enough, then the CPU becomes starved and the UDF is referred to as *memory-bound*. Load time is given by:

$$Load\ Time = \frac{Clock\ Speed \times Operand\ Size}{Bandwidth\ per\ Core} \tag{7}$$

For example, the load time for the `distance` UDF as shown in Table 2 computed on 32-bit floating-point (x, y) pairs using an Intel E5 processor with a 2.8GHz clock speed and 5.97GB/s memory bandwidth per core is calculated as follows: $3.75\ cycles = \frac{2.8GHz\times(2\times4B)}{5.97GB/s}$.

4.2 Optimizer

Tupleware's optimizer can apply a broad range of optimizations that occur on both a logical and physical level. We divide these optimizations into three categories.

High-Level: Tupleware utilizes well-known query optimization techniques, including predicate pushdown and join reordering. Additionally, our purely functional programming model allows for the integration of other

traditional optimizations from the programming language community. All high-level optimizations rely on metadata and algebra semantics, information that is unavailable to compilers, but are not particularly unique to Tupleware.

Low-Level: *Code generation* is the process by which compilers translate a high-level language (e.g., Tupleware's algebra) into an optimized low-level form (e.g., LLVM). As other work has shown [24], SQL query compilation techniques can harness the full potential of the underlying hardware, and Tupleware extends these techniques by applying them to the domain of complex analytics. As part of the translation process, Tupleware generates all of the data structure, control flow, synchronization, and communication code necessary to form a complete distributed program. Unlike other systems that use interpreted execution models, Volcano-style iterators, or remote procedure calls, Tupleware eliminates much associated overhead by compiling in these mechanisms. Tupleware also gains many compiler optimizations (e.g., SIMD vectorization, function inlining) "for free" by compiling workflows, but these optimizations occur at a much lower level than DBMSs typically consider.

Hybrid: Some systems incorporate DBMS and compiler optimizations separately, first performing algebraic transformations and then independently generating code based upon a fixed strategy. On the other hand, Tupleware combines an optimizable high-level algebra and statistics gathered by the Function Analyzer with the ability to dynamically generate code, enabling optimizations that would be impossible for either a DBMS or compiler alone. In particular, we consider (1) high-level algebra semantics, (2) metadata, and (3) low-level UDF statistics together to synthesize optimal code on a case-by-case basis. For instance, we are investigating techniques to generate different code for selections based upon the estimated selectivity and computational complexity of predicates.

4.3 Scheduler

The Scheduler determines how best to deploy a job given the available computing resources and physical data layout in the cluster. Most importantly, the Scheduler takes into account the optimum amount of parallelization for a given operation in a workflow. Operations on smaller datasets, for instance, may not benefit from massive parallelization due to the associated deployment overhead. Additionally, the Scheduler considers data locality to minimize data transfer between nodes.

5 Deployment

After program synthesis, the system now has a self-contained distributed program. Each distributed program contains all necessary communication and synchronization code, avoiding the overhead associated with external function calls. Tupleware takes a multitiered approach to distributed deployment, as shown in Figure 1. The system dedicates a single hyperthread on a single node in the cluster as the *Global Manager* (GM), which is responsible for global decisions such as the coarse-grained partitioning of the data across nodes and supervising the current stage of the workflow execution. In addition, we dedicate one thread per node as a *Local Manager* (LM). The LM is responsible for the fine-grained management of the local shared memory, as well as for transferring data between machines. The LM is also responsible for actually deploying compiled programs and does so by spawning new *executor threads* (E), which actually execute the workflow. During execution, these threads request data from the LM in an asynchronous fashion, and the LM responds with the data and an allocated result buffer.

Memory Management: Similar to DBMSs, Tupleware manages its own memory pool and tries to avoid memory allocations when possible. Therefore, the LM is responsible for keeping track of all active TupleSets and performing garbage collection when necessary. UDFs that allocate their own memory, though, are not managed by Tupleware's garbage collector. In addition, we avoid unnecessary object creations or data copying. For

instance, Tupleware often performs updates in-place if the data is not required in subsequent computations. Additionally, while the LM is idle, it can reorganize and compact the data, as well as free blocks of data that have already been processed.

Load Balancing: Tupleware's data request model is multitiered and pull-based, allowing for automatic load balancing with minimal overhead. Each of the executor threads requests data in small cache-sized blocks from the LM, and each LM in turn requests larger blocks of data from the GM. All remote data requests occur asynchronously, and blocks are requested in advance to mask transfer latency.

Fault Tolerance: As our experiments demonstrate, Tupleware can process gigabytes of data with subsecond response times, suggesting that checkpointing would do more harm than good. Extremely long-running jobs on the order of hours or days, though, might benefit from intermediate result recoverability. In these cases, Tupleware performs simple k-safe checkpoint replication. However, unlike other systems, Tupleware has a unique advantage: since we fully synthesize distributed programs, we can optionally add these recovery mechanisms on a case-by-case basis. If our previously described workflow analysis techniques determine that a particular job will have a long runtime, we combine that estimation with the probability of a failure (given our intimate knowledge of the underlying hardware) to decide whether to include checkpointing code.

6 Evaluation

We compared an early Tupleware prototype against Hadoop 2.4.0 and Spark 1.0.1 using a small cluster with high-end hardware. This cluster consisted of 10×`c3.8xlarge` instances with Intel E5-2680v2 processors (10 cores, 25MB Cache), 60GB RAM, 2 × 320GB SSDs, and 10 Gigabit*4 Ethernet.

6.1 Workloads and Data

The benchmarks included five common ML tasks. We implemented a consistent version of each algorithm across all systems with a fixed number of iterations and used synthetic datasets in order to test across a range of data characteristics (e.g., size, dimensionality, skew). All tasks operated on datasets of 1, 10, and 100GB in size. We recorded the total runtime of each algorithm after the input data was loaded into memory and parsed except in the case of Hadoop, which had to read from and write to HDFS on every iteration. For all iterative algorithms, we report the total time taken to complete 20 iterations. We now describe each ML task.

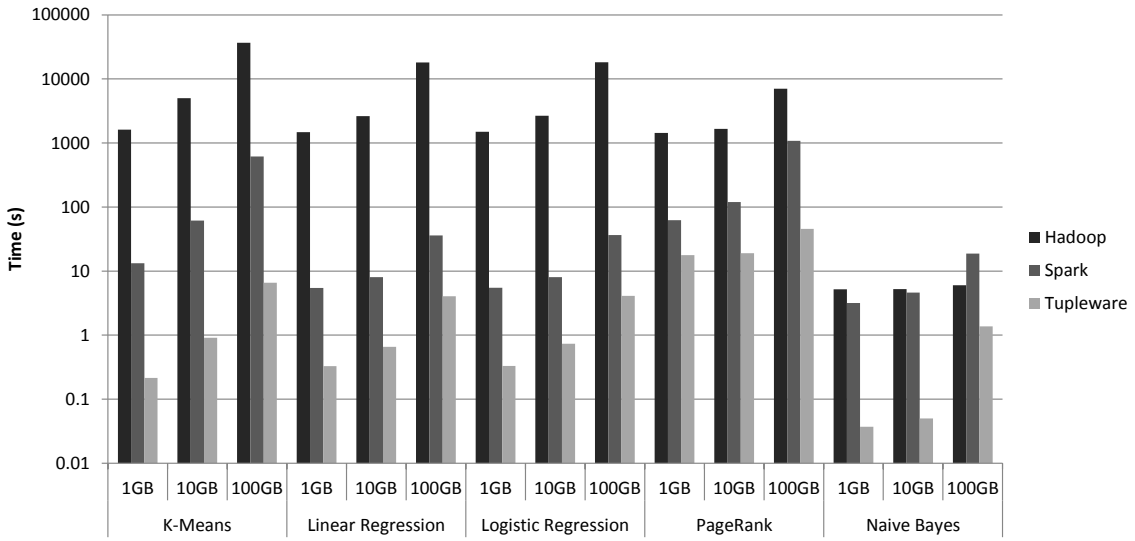
K-means: As described in Section 3.6, k-means is an iterative clustering algorithm that partitions a dataset into k clusters. Our test datasets were generated from four distinct centroids with a small amount of random noise.

Linear Regression: Linear regression produces a model by fitting a linear equation to a set of observed data points. We build the model using a parallelized batch gradient descent algorithm that computes updates locally on each worker from a disjoint subset of the dataset. These local updates are then averaged and applied to the model globally before the next iteration. The generated data had 1024 features.

Logistic Regression: Logistic regression attempts to find a hyperplane w that best separates two classes of data by iteratively computing the gradient and updating the parameters of w. We implemented logistic regression also with batch gradient descent on generated data with 1024 features.

PageRank: PageRank is an iterative link analysis algorithm that assigns a weighted rank to each page in a web graph to measure its relative significance. Based on the observation that important pages are likely to have more inbound links, search engines such as Google use the algorithm to order web search results.

Naive Bayes: A naive Bayes classifier is a conditional model that uses feature independence assumptions to assign class labels. Naive Bayes classifiers are used for a wide variety of tasks, such as spam filtering, text classification, and sentiment analysis. We trained a naive Bayes classifier on a generated dataset with 1024 features and 10 possible labels.



Algorithm	10×`c3.8xlarge`								
	Hadoop			**Spark**			**Tupleware**		
	1GB	10GB	100GB	1GB	10GB	100GB	1GB	10GB	100GB
K-means	1621.36	5023.64	36818.58	13.243	61.528	614.355	0.214	0.910	6.594
Linear Regression	1479.7	2622.68	18149.8	5.433	8.013	36.101	0.327	0.657	4.046
Logistic Regression	1497.66	2665.7	18200.54	5.507	8.030	36.630	0.329	0.734	4.107
PageRank	1438.44	1666.8	7019.52	62.348	119.854	1076.834	17.797	19.027	45.543
Naive Bayes	5.18	5.228	6.025	3.162	4.622	18.703	0.037	0.050	1.371

Figure 5: Distributed benchmark results and runtimes (in seconds).

6.2 Discussion

As shown in Figure 5, Tupleware outperforms Hadoop by up to three orders of magnitude and Spark by up to two orders of magnitude for the tested ML tasks on a small cluster of high-end hardware.

Tupleware is able to significantly outperform Hadoop because of the substantial I/O overhead required for materializing intermediate results to HDFS between iterations. On the other hand, Tupleware is able to cache intermediate results in memory and performs hardware-level optimizations to improve CPU efficiency. For this reason, we measure the greatest speedups over Hadoop on k-means, linear and logistic regression, and PageRank, whereas the performance difference for Naive Bayes is not as pronounced. Furthermore, Hadoop's simple API is not intended for complex analytics and the system is not designed to optimize workflows for single-node performance.

Spark performs better than Hadoop for the iterative algorithms because it allows users to cache the working set in memory, eliminating the need to materialize intermediate results to disk. Additionally, Spark offers a richer API that allows the runtime to pipeline operators, further improving data locality and overall performance. For CPU-intensive ML tasks such as k-means, though, Tupleware is able to significantly outperform Spark by

synthesizing distributed programs and employing low-level code generation optimizations. Furthermore, Tupleware's unique frontend and Context variables described in Section 3 provide efficient, globally-distributed shared state for storing and updating a model. The benefit of storing the model as a Context variable is most noticeable in our naive Bayes experiment due to the fact that each machine can directly update a local copy of the model instead of performing an expensive shuffle operation.

7 Related Work

Tupleware's unique design allows the system to highly optimize complex analytics tasks. While other systems have looked at individual components, Tupleware collectively addresses how to (1) easily and concisely express complex analytics workflows, (2) synthesize self-contained distributed programs optimized at the hardware level, and (3) deploy tasks efficiently on a cluster.

7.1 Programming Model

Numerous extensions have been proposed to support iteration and shared state within MapReduce [10, 18, 7], and some projects (e.g., SystemML [19]) go a step further by providing a high-level language that is translated into MapReduce tasks. Conversely, Tupleware natively integrates iterations and shared state to support this functionality without sacrificing low-level optimization potential. Other programming models, such as FlumeJava [13], Ciel [27], and Piccolo [29] lack the low-level optimization potential that Tupleware's algebra provides.

DryadLINQ [35] is similar in spirit to Tupleware's frontend and allows users to perform relational transformations directly in any .NET host language. Compared to Tupleware, though, DryadLINQ cannot easily express updates to shared state and requires an external driver program for iterative queries, which precludes any cross-iteration optimizations.

Scope [12] provides a declarative scripting language that is translated into distributed programs for deployment in a cluster. However, Scope primarily focuses on SQL-like queries against massive datasets rather than supporting complex analytics workflows.

Tupleware also has commonalities with the programming models proposed by Spark [36] and Stratosphere [21]. These systems have taken steps in the right direction by providing richer APIs that can supply an optimizer with additional information about the workflow, thus permitting standard high-level optimizations. In addition to these more traditional optimizations, Tupleware's algebra is designed specifically to enable low-level optimizations that target the underlying hardware, as well as to efficiently support distributed shared state.

7.2 Code Generation

Code generation for query evaluation was proposed as early as System R [8], but this technique has recently gained popularity as a means to improve query performance for in-memory DBMSs [30, 24]. Both HyPer [22] and VectorWise [37] propose different optimization strategies for query compilation, but these systems focus on SQL and do not optimize for UDFs. LegoBase [23] includes a query engine written in Scala that generates specialized C code and allows for continuous optimization, but LegoBase also concentrates on SQL and does not consider ML or UDFs.

DryadLINQ compiles user-defined workflows into executables using the .NET framework but applies only traditional high-level optimizations. Similarly, Tenzing [14] and Impala [2] are SQL compilation engines that also focus on simple queries over large datasets.

OptiML [33] offers a Scala-embedded, domain-specific language used to generate execution code that targets specialized hardware (e.g., GPUs) on a single machine. Tupleware on the other hand provides a general, language-agnostic frontend used to synthesize LLVM-based distributed executables for deployment in a cluster.

7.3 Single-Node Frameworks

BID Data Suite [11] and Phoenix [34] are high performance single-node frameworks targeting general analytics, but these systems cannot scale to multiple machines or beyond small datasets. Scientific computing languages like R [6] and Matlab [5] have these same limitations. More specialized systems (e.g., Hogwild! [31]) provide highly optimized implementations for specific algorithms on a single machine, whereas Tupleware is intended for general computations in a distributed environment.

8 Conclusion

Advanced analytics workloads, in particular distributed ML, have become commonplace for a wide range of users. However, instead of targeting the hardware to which most of these users have access, existing frameworks are designed primarily for large cloud deployments with thousands of commodity machines. This paper described our vision for Tupleware, a new analytics system geared towards compute-intensive, in-memory analytics on small clusters. Tupleware combines ideas from the database and compiler communities to create a user-friendly yet highly efficient end-to-end data analysis solution. Our preliminary experiments demonstrated that our approach can achieve speedups of up to several orders of magnitude for common ML tasks.

References

[1] Apache hadoop. `http://hadoop.apache.org`.

[2] Cloudera impala. `http://www.cloudera.com/content/cloudera/en/products-and-services/cdh/impala.html`.

[3] Lists of instruction latencies, throughputs and micro-operation breakdowns for intel, amd and via cpus. `http://www.agner.org/optimize/instruction_tables.pdf`.

[4] Mapreduce: A major step backwards. `http://homes.cs.washington.edu/~billhowe/mapreduce_a_major_step_backwards.html`.

[5] Matlab. `http://www.mathworks.com/products/matlab/`.

[6] R project. `http://www.r-project.org/`.

[7] S. Alsubaiee, Y. Altowim, H. Altwaijry, A. Behm, V. R. Borkar, Y. Bu, M. J. Carey, R. Grover, Z. Heilbron, Y.-S. Kim, C. Li, N. Onose, P. Pirzadeh, R. Vernica, and J. Wen. Asterix: An open source system for "big data" management and analysis. *PVLDB*, 5(12):1898–1901, 2012.

[8] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, J. Gray, W. F. K. III, B. G. Lindsay, R. A. Lorie, J. W. Mehl, T. G. Price, G. R. Putzolu, M. Schkolnick, P. G. Selinger, D. R. Slutz, H. R. Strong, P. Tiberio, I. L. Traiger, B. W. Wade, and R. A. Yost. System r: A relational data base management system. *IEEE Computer*, 12(5):42–48, 1979.

[9] V. R. Borkar, Y. Bu, M. J. Carey, J. Rosen, N. Polyzotis, T. Condie, M. Weimer, and R. Ramakrishnan. Declarative systems for large-scale machine learning. *IEEE Data Eng. Bull.*, 35(2):24–32, 2012.

[10] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. Haloop: Efficient iterative data processing on large clusters. *Proc. VLDB Endow.*, 3(1-2):285–296, Sept. 2010.

[11] J. Canny and H. Zhao. Big data analytics with small footprint: Squaring the cloud. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '13, pages 95–103, New York, NY, USA, 2013. ACM.

[12] R. Chaiken, B. Jenkins, P.-A. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. Scope: Easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow.*, 1(2):1265–1276, Aug. 2008.

[13] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum. Flumejava: easy, efficient data-parallel pipelines. In B. G. Zorn and A. Aiken, editors, *PLDI*, pages 363–375. ACM, 2010.

[14] B. Chattopadhyay, L. Lin, W. Liu, S. Mittal, P. Aragonda, V. Lychagina, Y. Kwon, and M. Wong. Tenzing a sql implementation on the mapreduce framework. In *Proceedings of VLDB*, pages 1318–1327, 2011.

[15] Y. Chen, S. Alspaugh, and R. Katz. Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads. *Proc. VLDB Endow.*, 5(12):1802–1813, Aug. 2012.

[16] T. Cruanes, B. Dageville, and B. Ghosh. Parallel sql execution in oracle 10g. SIGMOD '04, pages 850–854, New York, NY, USA, 2004. ACM.

[17] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[18] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox. Twister: a runtime for iterative mapreduce. In *HPDC*, pages 810–818, 2010.

[19] A. Ghoting, R. Krishnamurthy, E. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian, and S. Vaithyanathan. Systemml: Declarative machine learning on mapreduce. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, ICDE '11, pages 231–242, Washington, DC, USA, 2011. IEEE Computer Society.

[20] B. Graham and M. R. Rangaswami. Do you hadoop? a survey of big data practitioners. Sandy Hill Group, 2013.

[21] F. Hueske, M. Peters, A. Krettek, M. Ringwald, K. Tzoumas, V. Markl, and J.-C. Freytag. Peeking into the optimization of data flow programs with mapreduce-style udfs. In C. S. Jensen, C. M. Jermaine, and X. Zhou, editors, *ICDE*, pages 1292–1295. IEEE Computer Society, 2013.

[22] A. Kemper, T. Neumann, J. Finis, F. Funke, V. Leis, H. Mühe, T. Mühlbauer, and W. Rödiger. Processing in the hybrid oltp & olap main-memory database system hyper. *IEEE Data Eng. Bull.*, 36(2):41–47, 2013.

[23] Y. Klonatos, C. Koch, T. Rompf, and H. Chafi. Building efficient query engines in a high-level language. *PVLDB*, 7(10):853–864, 2014.

[24] K. Krikellas, S. Viglas, and M. Cintra. Generating code for holistic query evaluation. In *ICDE*, pages 613–624, 2010.

[25] C. Lattner and V. S. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88, 2004.

[26] K.-H. Lee, Y.-J. Lee, H. Choi, Y. D. Chung, and B. Moon. Parallel data processing with mapreduce: a survey. *SIGMOD Record*, 40(4):11–20, 2011.

[27] D. G. Murray, M. Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy, and S. Hand. Ciel: A universal execution engine for distributed data-flow computing. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 113–126, Berkeley, CA, USA, 2011. USENIX Association.

[28] A. Nadkarni and L. DuBois. Trends in enterprise hadoop deployments. IDC, 2013.

[29] R. Power and J. Li. Piccolo: Building fast, distributed programs with partitioned tables. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–14, Berkeley, CA, USA, 2010. USENIX Association.

[30] J. Rao, H. Pirahesh, C. Mohan, and G. M. Lohman. Compiled query execution engine using jvm. In *ICDE*, page 23, 2006.

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

[32] A. Rowstron, D. Narayanan, A. Donnelly, G. O'Shea, and A. Douglas. Nobody ever got fired for using hadoop on a cluster. In *Proceedings of the 1st International Workshop on Hot Topics in Cloud Data Processing*, HotCDP '12, pages 2:1–2:5, New York, NY, USA, 2012. ACM.

[33] A. K. Sujeeth, H. Lee, K. J. Brown, H. Chafi, M. Wu, A. R. Atreya, K. Olukotun, T. Rompf, and M. Odersky. Optiml: an implicitly parallel domainspecific language for machine learning. In *in Proceedings of the 28th International Conference on Machine Learning, ser. ICML*, 2011.

[34] J. Talbot, R. M. Yoo, and C. Kozyrakis. Phoenix++: Modular mapreduce for shared-memory systems. In *Proceedings of the Second International Workshop on MapReduce and Its Applications*, MapReduce '11, pages 9–16, New York, NY, USA, 2011. ACM.

[35] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, pages 1–14, 2008.

[36] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

[37] M. Zukowski, M. van de Wiel, and P. A. Boncz. Vectorwise: A vectorized analytical dbms. In *ICDE*, pages 1349–1350, 2012.

Cumulon: Cloud-Based Statistical Analysis from Users' Perspective

Botong Huang
Department of Computer Science
Duke University
bhuang@cs.duke.edu

Nicholas W.D. Jarrett
Department of Statistical Science
Duke University
nwj2@stat.duke.edu

Shivnath Babu
Department of Computer Science
Duke University
shivnath@cs.duke.edu

Sayan Mukherjee
Department of Statistical Science
Duke University
sayan@stat.duke.edu

Jun Yang
Department of Computer Science
Duke University
junyang@cs.duke.edu

Abstract

Cumulon *is a system aimed at simplifying the development and deployment of statistical analysis of big data on public clouds.* Cumulon *allows users to program in their familiar language of matrices and linear algebra, without worrying about how to map data and computation to specific hardware and software platforms. Given user-specified requirements in terms of time, money, and risk tolerance,* Cumulon *finds the optimal implementation alternatives, execution parameters, as well as hardware provisioning and configuration settings—such as what type of machines and how many of them to acquire.* Cumulon *also supports clouds with auction-based markets: it effectively utilizes computing resources whose availability varies according to market conditions, and suggests best bidding strategies for such resources. This paper presents an overview of* Cumulon *and the challenges encountered in building this system.*

1 Introduction

The ubiquity of the *cloud* today presents an exciting opportunity to make big-data analytics more accessible than ever before. Users who want to perform statistical analysis on big data are no longer limited to those at big Internet companies or HPC (High-Performance Computing) centers. Instead, they range from small business owners, to social and life scientists, and to legal and journalism professionals, many of whom have limited computing expertise but are now beginning to see the potential of the commoditization of computing resources. With publicly available clouds such as Amazon EC2, Microsoft Azure, and Google Cloud, users can rent a great variety of computing resources instantaneously, and pay only for their use, without worrying about acquiring, hosting, and maintaining physical hardware.

Unfortunately, many users still find it frustratingly difficult to use the cloud for any non-trivial statistical analysis of big data. ***Developing*** efficient statistical analysis programs requires tremendous expertise and effort.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

Most statisticians would much prefer programming in languages familiar to them, such as R and MATLAB, where they can think and code naturally in terms of matrices and linear algebra. However, scaling programs written in these languages to bigger data is hard. The straightforward solution of getting a bigger machine with more memory and CPU quickly becomes prohibitively expensive as data volume continues to grow. Running on an HPC cluster requires hardware resources and parallel programming expertise that few have access to. Although the cloud has made it considerably easier than before to tap into the power of parallel processing using commodity hardware, popular cloud programming platforms, such as *Hadoop*, still require users to think and code in low-level, platform-specific ways. Emerging libraries for statistical analysis and machine learning on these platforms help alleviate the difficulty somewhat, but in many cases there exists no library tuned for the specific problem, platform, and budget at hand, so users must resort to extensive retooling and manual tuning. To keep up with constant innovations in data analysis, systems that support rapid development of brand new computational procedures are sorely needed.

Deploying statistical analysis programs in the cloud is also difficult. Users face a maddening array of choices, including hardware provisioning (e.g., *"should I get five powerful machines of this type, or a dozen of these cheaper, less powerful ones?"*), configuration (e.g., *"how do I set the numbers of map and reduce slots in Hadoop?"*), and execution parameters (e.g., *"what should be the size of each parallel task?"*). Furthermore, these deployment decisions may be intertwined with development—if a program is written in a low-level, non-"declarative" manner, its implementation alternatives and deployment decisions will affect each other. Finally, the difficulty of decision making is compounded by the emergence of auction-based markets, such as Amazon *spot instances*, where users can bid for computing resources whose availability is subject to market conditions. Current systems offer little help to users in making such decisions.

Many users have long been working with data successfully in their domains of expertise—be they statisticians in biomedical and policy research groups, or data analysts in media and marketing companies—but now they find it difficult to extend their success to bigger data because of lack of tools that can spare them from low-level development and deployment details. Moreover, while the cloud has greatly widened assess to computing resources, it also makes the risk of mistakes harder to overlook—wrong development and deployment decisions now have a clear price tag (as opposed to merely wasted cycles on some computers). Because of these issues, many potential users have been reluctant to move their data analysis to the cloud.

Cumulon is an ongoing project at Duke University aimed at simplifying the development and deployment of statistical analysis in the cloud. When developing such programs, we want users to think and code in a high-level language, without being concerned about how to map data and computation onto specific hardware and software platforms. Currently, *Cumulon* targets matrix computation workloads. Many statistical data analysis methods (or computationally expensive components thereof) can be expressed naturally with matrices and linear algebra. For example, consider *singular value decomposition* (*SVD*) of matrices, which has extensive applications in statistical analysis. In the randomized algorithm of [18] for computing an approximate SVD, the first (and most expensive) step, which we shall refer to as RSVD-1, involves a series of matrix multiplies. Specifically, given an $m \times n$ input matrix $\mathbf{A}$, this step uses an $l \times m$ randomly generated matrix $\mathbf{G}$ whose entries are i.i.d. Gaussian random variables of zero mean and unit variance, and computes $\mathbf{G} \times (\mathbf{A} \times \mathbf{A})^k \times \mathbf{A}$.

When deploying such programs, users should be able to specify their objectives and constraints in straightforward terms—time, money, and risk tolerance. Then, *Cumulon* will present users with the best "plans" meeting these requirements. A plan encodes choices of not only implementation alternatives and execution parameters, but also cluster resource and configuration parameters. For example, Figure 1 shows the costs[1] of best plans for RSVD-1 as we vary the constraint on expected completion time. In general, the best plans differ across points in this figure; choosing them by hand would have been tedious and difficult. The figure reveals a clear trade-off between completion time and cost, as well as the relative cost-effectiveness of various machine types for the

[1]To simplify the interpretation of results in this paper, we do not follow Amazon EC2's practice of rounding usage time to full hours when computing the costs reported here (though it is straightforward for *Cumulon* to use that pricing policy).

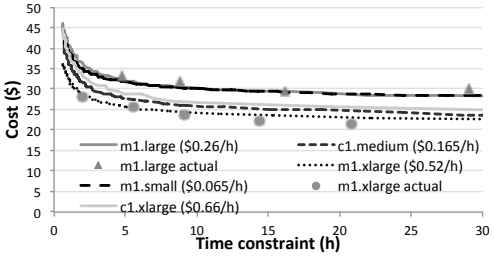


Figure 1: (From [10]) Costs of the optimal deployment plans for RSVD-1 (with $l = 2{,}000$, $m = n = 200{,}000$, $k = 5$) using different Amazon EC2 machine types under different time constraints. Curves are predicted; costs of actual runs for sample data points are shown for comparison.

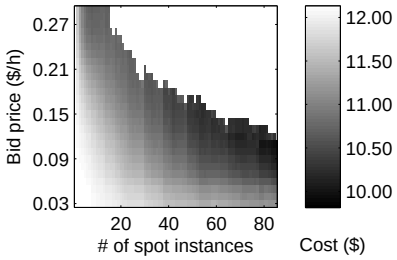


Figure 2: (From [11]) Estimated expected costs of the optimal plans for RSVD-1 (with $l = 2{,}000$, $m = n = 160{,}000$, $k = 5$), as we vary the bid price and number of spot instances to bid for. The cost is shown using intensity, with darker shades indicating lower costs. In the upper-right region, plans with the given bidding strategies fail to meet the user-specified risk tolerance; therefore, this region shows the baseline cost of the optimal plan with no bidding. All machines are of type `c1.medium`.

given program, making it easier for users to make informed decisions.

Cumulon is also helpful when working with an auction-based market of computing resources. For example, to encourage utilization of spare capacity, Amazon lets users bid for *spot instances*, whose *market prices* change dynamically but are often much lower than the regular fixed prices. Users pay the market price of a spot instance for each time unit when it is in use; however, as soon as the market price exceeds the bid price, the spot instance will be taken away.[2] For example, consider again RSVD-1.[3] It takes a baseline cost of $12.12 to run under 28 hours, using 3 machines of type `c1.medium` at the fixed price of $0.145 per hour. Taking advantage of additional spot instances, we can potentially complete faster and end up paying less overall than the baseline. Because of the uncertainty in future market prices, *Cumulon* lets users specify a risk tolerance—e.g., *"with probability no less than* 0.9*, the overall cost will not exceed the baseline by more than* 5%*."* Figure 2 shows the expected cost of optimal plans for RSVD-1 that meet the given risk tolerance, under various bidding strategies. Using information in this figure, *Cumulon* can recommend that user bid for additional 77 `c1.medium` spot instances at $0.11 per hour each (versus the current market price of $0.02), which would reduce the expected overall cost to $10.00 while staying within the user's risk tolerance. Later in this paper, we will discuss how *Cumulon* makes such recommendations.

Challenges An analogy between *Cumulon* and a database system is useful. Much like a database system, *Cumulon* starts with a *logical plan* representing the input program, expressed in terms of well-understood matrix primitives—such as multiply, add, transpose, etc.—instead of relational algebra operators. Then, *Cumulon* applies cost-based optimization to find optimal *physical plans* (or simply *plans*, if the context is clear) implementing the given logical plan. Despite this high-level similarity, however, a number of unique challenges take *Cumulon* well beyond merely applying database techniques to cloud-based matrix computation.

To begin, there is a large and interesting design space for storage and execution engines in this setting. We have three pillars to build on: 1) the database approach, whose pipelined, operator-based execution makes it easy to discern semantics and process out-of-core data; 2) the data-parallel programming approach, represented by

[2] Amazon EC2 actually does not charge for partial hour of usage of spot instances if they are taken away (as opposed to terminated voluntarily by users). This policy is specific to Amazon and can lead to some rather interesting bidding schemes. To avoid making our results too specific to Amazon, we always consider fractional hours when computing costs in this paper and make no special case for spot instances. (Again, *Cumulon* can readily support the actual Amazon policy instead.)

[3] Please note that there are minor differences between the workload settings and prices in the examples and figures of this paper, because the results were originally reported in different papers [10, 11] and obtained at times when Amazon charged different prices.

MapReduce [6], whose simplicity allows it to scale out to big data and large commodity clusters common to the cloud; and 3) the HPC approach, which offers highly optimized libraries for in-core matrix computation. Each approach has its own strengths and weaknesses, and none meets all requirements of *Cumulon*. When developing *Cumulon*, we made a conscious decision to avoid "reinventing the wheel," and to leverage the popularity of existing cloud programming platforms such as Hadoop. Unfortunately, MapReduce has fundamental limitations when handling matrix computation. Therefore, *Cumulon* must incorporate elements of database- and HPC-style processing, but questions remain: How effectively can we put together these elements within the confines of existing platforms? Given the fast-evolving landscape of cloud computing, how can we reduce *Cumulon*'s dependency on specific platforms?

Additional challenges arise when supporting *transient nodes*, which refer to machines acquired through bidding, such as Amazon spot instances, with dynamic, market-based prices and availability. Getting lots of cheap, transient nodes can pay off by reducing the overall completion time, but losing them at inopportune times will lead to significant loss of work and higher overall cost. Although current cloud programming platforms handle node failures, most of them do not expect simultaneous departures of a large number (and easily the majority) of nodes as a common occurrence. Any efforts to mitigate the loss of work, such as copying intermediate results out of transient nodes, must be balanced with the associated overhead and potential bottlenecks, through a careful cost-benefit analysis. *Cumulon* needs storage and execution engines capable of handling massive node departures and supporting intelligent policies for preserving and recovering work on transient nodes.

Optimization is another area ripe with new challenges. Compared with traditional query optimization, *Cumulon*'s plan space has more dimensions—from settings of hardware provisioning and software configuration, to strategies for bidding for transient nodes and preserving their work. *Cumulon* formulates its optimization problem differently from database systems, as it targets user-centric metrics of time and money, as opposed to system-centric metrics such as throughput. Uncertainty also plays a more important role in *Cumulon*, because the cloud brings more uncertainty to cost estimation, and because *Cumulon* is user-facing: while good average-case performance may be acceptable from a system's perspective, cost variability must also be considered from a user's perspective. Therefore, *Cumulon* lets users specify their risk tolerance in optimization, allowing, for example, only plans that stay within budget with high certainty.

To support this uncertainty-aware, cost-based optimization, *Cumulon* faces many challenges in cost estimation that are distinct from database systems. While cardinality estimation is central to query optimization, it is less of an issue for *Cumulon*, because the sizes of matrices (including intermediate results) are usually known at optimization time. On the other hand, modeling of future market prices and modeling of placement of intermediate results become essential in predicting the costs of plans involving transient nodes. *Cumulon* needs to build uncertainty into its modeling, not only to support the risk tolerance constraint in optimization, but also to help understand the overall behavior of execution. For example, quantifying the variance among the speeds of individual threads of parallel execution improves the estimation of overall completion time.

In the rest of this paper, we give an overview of how *Cumulon* deals with these challenges, and then conclude with a discussion of related and future work. For further details on *Cumulon*, please see [10, 11].

2 Storage and Execution

Cumulon provides an abstraction for distributed storage of matrices. Matrices are stored and accessed by *tiles*, where each tile is a submatrix of fixed (but configurable) dimension. For large matrices, we choose a large tile size (e.g., 32MB) to amortize the overhead of storage and access, and to enable effective compression.

At a high level, *Cumulon*'s execution model has much in common with popular data-parallel programming platforms. A *Cumulon* program executes as a workflow of *jobs*. Each job reads and writes a number of matrices; input and output matrices must be disjoint. Dependent jobs execute in a serial order. Each job executes as multiple independent *tasks* that do not communicate with each other. All tasks within the job run identical code,

but read different (possibly overlapping) parts of the input matrices, and write disjoint parts of output matrices. Data are passed between jobs only through the global, distributed storage.

Cumulon configures each available *node* (machine) into a number of *slots*, and schedules each slot to execute one task at a time. If a job consists of more tasks than slots, it will take multiple waves to finish. Elements of both database- and HPC-style processing are incorporated into task execution. *Cumulon* executes each task as a directed acyclic graph of physical operators in a pipelined fashion, much like the iterator-based execution in database systems. However, the unit of data passing between physical operators is much bigger than in database systems—we pass tiles instead of elements, to reduce overhead, and to be able to use the highly tuned BLAS library on submatrices. Physical operators can choose to batch multiple tiles in memory to create larger submatrices, which increase CPU utilization at the expense of memory consumption. For example, *Cumulon*'s matrix multiply operator provides a *streaming granularity* parameter that can be adjusted (automatically by *Cumulon*'s optimizer) to achieve the best trade-off for a given hardware configuration.

MapReduce No, Hadoop Yes It is worth pointing out that *Cumulon* does not follow the popular MapReduce model. In *Cumulon*, different tasks can read overlapping parts of the input data, directly from the distributed storage; there is no requirement of disjoint input partitioning or *shuffle*-based data passing as in MapReduce. This flexibility is crucial to efficient matrix computation, which often exhibits access patterns awkward to implement with MapReduce. In [10], we have demonstrated that *Cumulon*'s execution model enables far more efficient implementation choices of matrix workloads than approaches based on MapReduce.

For example, consider multiplying two big matrices $\mathbf{A}$ and $\mathbf{B}$. On a high level, virtually all efficient parallel matrix multiply algorithms work by dividing $\mathbf{A}$ and $\mathbf{B}$ into submatrices of appropriate dimensions: first, pairs of submatrices from the two inputs are multiplied in parallel; second, the results of the multiplies are grouped (by the output submatrix position they contribute to) and summed. In general, one input submatrix may be needed by multiple tasks in the first step. Under MapReduce, we would have to use a map phase to replicate the input submatrices and shuffle them to the reduce phase to be multiplied. Then, we need another round of map and reduce for the summation. In both steps, the map phase performs no useful computation. There are workarounds that address some aspects of this inefficiency by choosing submatrices consisting of entire rows and columns, but such submatrix dimensions are often suboptimal, leading to low compute-to-I/O ratios. In contrast, *Cumulon* supports fully flexible choices of submatrix dimensions, using simply one job for submatrix multiplies and one for summation, both performing useful work. *Cumulon* can further reduce overhead by folding the second job into a subsequent one that uses the result of $\mathbf{A} \times \mathbf{B}$, an optimization that we will come back to in Section 3.

Despite MapReduce's inadequacy, we still want to leverage the popularity of MapReduce-based platforms. Hadoop was an obvious choice when we started the *Cumulon* project, because it already had a healthy, growing ecosystem of developers, users, and service providers. Although *Cumulon*'s storage and execution models differ from MapReduce, we managed to implement *Cumulon* on top of Hadoop and HDFS without changing their internals. Specifically, every *Cumulon* job is implemented as a map-only job in Hadoop that can access specific tiles directly through a distributed tile storage implemented on HDFS. For example, Figure 3 illustrates how *Cumulon* implements $\mathbf{A} \times \mathbf{B} + \mathbf{C}$ on Hadoop and HDFS. Here, the summation step of $\mathbf{A} \times \mathbf{B}$ has been folded into the job that adds $\mathbf{C}$. Building on top of Hadoop and HDFS, *Cumulon* is able to inherit their features of scalability, elasticity, and failure handling; it is also easy to support complex workflows that combine *Cumulon*-powered matrix computation with regular Hadoop jobs for data wrangling and processing.

Finally, we note that *Cumulon*'s storage and execution models are simple and generic by design, so we can potentially implement new instances of *Cumulon* on top of other platforms such as *Spark* [26] and *Dryad* [12].

Support for Transient Nodes Making effective use of transient nodes requires attention to multiple points throughout *Cumulon*. For storing intermediate results, *Cumulon* does not assume that a distributed storage service (such as Amazon EBS or S3) exists independently of the provisioned nodes. Local storage attached

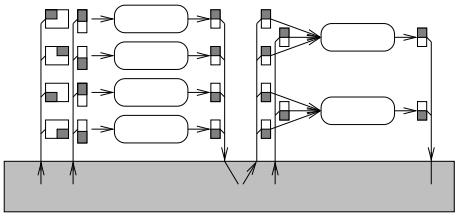


Figure 3: (From [10]) *Cumulon*'s map-only Hadoop jobs for $\mathbf{A} \times \mathbf{B} + \mathbf{C}$. Shaded blocks represent submatrices. $\{\mathbf{T}_k\}$ is the collection of results of from submatrix multiplies, yet to be grouped and summed.

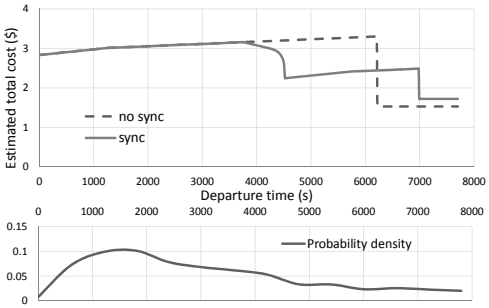


Figure 4: (From [11]) *Top*: Estimated costs of two plans (with and without *sync*) for a chain of five matrix multiplies. The *sync* job is applied to result of the third multiply. All matrices have dimension $30{,}000 \times 30{,}000$. There are 3 primary and 10 transient nodes, all of type `c1.medium`. *Bottom*: Probability density function for the distribution of the departure time of transient nodes, with bid price of $\$0.05$ and current market price of $\$0.02$. The model is based on the historical prices of Amazon spot instances.

directly to the nodes are cheaper and faster, but a sudden departure of transient nodes will lead to loss of valuable intermediate results. To ensure progress of execution, *Cumulon* provisions a set of *primary nodes*, which are regular machines acquired at fixed prices whose availability is not subject to market conditions. A naive strategy for preventing loss of work is for the transient nodes to write all intermediate results to primary node storage, but doing so is costly and can overwhelm the primary nodes (especially since we often use more transient nodes than primary ones). In general, it is difficult for the storage layer to decide intelligently what data to save, because that decision depends on when the data will be used, when the transient nodes are expected to depart, and how expensive it is to recompute the data. Therefore, *Cumulon* takes a cross-layer approach. In the storage layer, *Cumulon* caches data opportunistically—whenever a primary node accesses data from a transient node (or vice versa). Additionally, *Cumulon*'s execution engine supports a *sync* operator, which, for a given matrix, copies any tiles not already stored by the primary nodes from the transient nodes in a distributed fashion. *Cumulon*'s optimizer decides when and for which matrices *sync* is needed.

Support for transient nodes also necessitates support for recovery and heterogeneous clusters. Recovery is needed following the departure of the transient nodes. During recovery, *Cumulon* automatically identifies missing tiles required to complete the execution, and uses the lineage information inferred from the execution plan to recompute these tiles. Heterogeneity is needed because *Cumulon* may acquire transient nodes of a type different from that of the primary nodes (and may do so after the execution starts). It turns out that the optimal size of a task depends on what hardware it runs on. Therefore, *Cumulon* has an online task scheduler that dynamically assigns tasks of different sizes according to the available node types. The scheduler maintains a map for each output matrix, and, upon request to assign a task to a node, splits off a region of responsibility whose dimension best matches the optimal task size for the node type.

Again, we are able to implement support for transient nodes in *Cumulon* on top of Hadoop and HDFS. We use a custom Hadoop scheduler. The distribution tile storage runs on top of two HDFS instances—one involving all primary nodes and the other involving all transient nodes—and caches data between the instances upon access. The dual-HDFS approach is clean and resilient against the massive departure of all transient nodes; using a single HDFS would require non-trivial modification to its replication logic in order to provide a similar level of resiliency. The *sync* operator and recovery phase are both implemented as Hadoop jobs.

Figure 4 (focus on the top figure for now) illustrates the benefit of transient nodes and importance of *sync* for a simple workload consisting of a series of five matrix multiplies. The figure shows how the costs of two different plans (with and without *sync*) vary as the departure time of the transient nodes changes. Here, *sync* applies to the result of the third multiply. The two curves start from the same point on the left, where the cost

reflects the baseline cost when no transient machines are used (here, they depart immediately and incur no extra cost). If both plans run to completion without transient nodes departing, their costs (shown on the right where the curves become flat) would be much lower than the baseline, showing the benefit of transient nodes (in that case, the *sync* plan actually takes longer and costs slightly more than the no-*sync* plan, because of the overhead of *sync* itself). On the other hand, if the transient nodes depart during its execution, the no-*sync* plan would cost more than the baseline, as it loses valuable work done on the transient nodes. Fortunately, *sync* comes to the rescue. If the transient nodes depart between ≈ 4000s (when *sync* starts) and ≈ 6100s into the execution (when the no-*sync* plan would complete), the *sync* plan will cost less than the no-*sync* plan. As long as the departure occurs after ≈ 4200s (when *sync* is almost complete), the *sync* plan will cost less than the baseline. Of course, the departure probability of transient nodes is not uniform over time; we must account for this uncertainty when comparing these plans and the baseline. We will revisit this example when discussing optimization in Section 3.

3 Optimization

A simpler version of the optimization problem, used by our first iteration of *Cumulon* in [10] for plans that use no transient nodes, ignores risk tolerance. The resulting optimization is bi-criteria, involving time and money. One possible formulation is to find the plan with the lowest expected monetary cost among those expected to complete under a user-specified time constraint. The optimizer is given the logical plan representing the program, as well as input data characteristics (such as dimensions and sparsity of matrices). Thanks to the nature of matrix computation, performance uncertainty is manageable for many workloads, so this simple approach is practical.

Once we consider transient nodes, however, the degree of uncertainty becomes significant enough to affect user decisions. Therefore, we extend the optimization problem as follows. Let C denote be the expected monetary cost of the best plan (without using transient nodes) obtained by solving the simpler version of the optimization problem under the user-specified time constraint. *Cumulon* then looks for the plan (with possible use of transient nodes) having the lowest expected monetary cost, subject to the constraint that the actual cost of the plan exceeds δC with probability less than ϵ; here, (δ, ϵ) is the user-specified risk tolerance.

We clarify what we mean by a "plan with possible use of transient nodes" (although we omit the formal definition of the plan space here). First, because the optimization decision is based on the current market condition, the plan is intended to begin executing now, with a set of primary nodes. Optionally, the plan can bid for a number of transient nodes at a particular price, either immediately or at some point later during the execution. If the bid is successful, the plan will execute on both primary and transient nodes, with *sync* jobs at predetermined points in the execution. If the transient nodes depart before the execution completes, the primary nodes enter a recovery phase to bring the execution to a state consistent with one that would have been achieved by completing all jobs started by the partial execution on the primary nodes alone. Normal execution then resumes. The optimizer is responsible for choosing the primary node cluster, the transient node cluster (if any) and its bid price and time, as well as the execution plan (with *sync* jobs if applicable).

Note that in the above, we consider only a *single* round of decision about whether, when, and how to bid for transient nodes. The benefit of optimizing further ahead into the future (e.g., whether to bid for more transient nodes in additional rounds) is unclear, because market uncertainty widens quickly over a longer time horizon. A better approach, which we are actively investigating for *Cumulon*, is to optimize and adapt *online* as we continue to monitor execution progress and market conditions. With some extension, the single-round optimization problem above could serve as a useful building block for the online optimizer—at any point during execution, we can solve this problem for the remaining jobs to see whether we should wait or bid for transient nodes now. Another extension orthogonal to the use of transient nodes is cluster switching: we can partition a program into parts to be handled by different clusters best suited for their execution, with necessary data transfers between the clusters. *Cumulon* already considers and supports switching of the primary node cluster [10].

In the following, we briefly discuss cost estimation techniques and optimization algorithms in *Cumulon*.

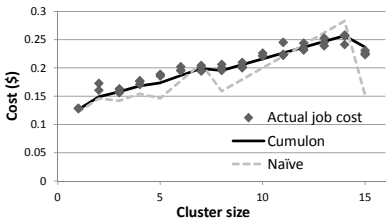


Figure 5: (From [10]) Actual job costs vs. predictions from the naive method and *Cumulon*. The job multiplies two dense matrices of sizes $12{,}000 \times 18{,}000$ and $18{,}000 \times 30{,}000$, and consists of 30 tasks. Machine type is `c1.medium`, with 2 slots per node.

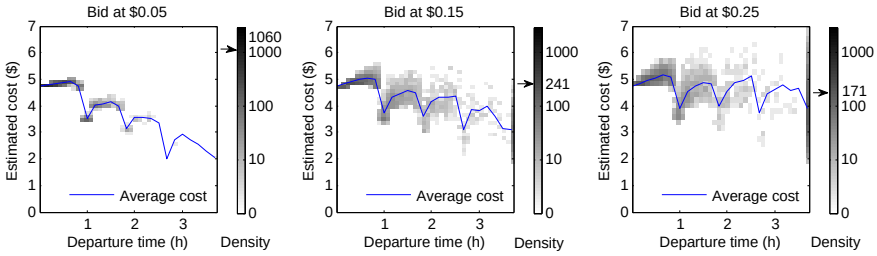


Figure 6: (From [11]) Effect of bidding price on the distribution of plan cost. Here, the workload consists of 12 jobs implementing two iterations of Gaussian non-negative matrix factorization. The baseline plan runs on a 3-node `c1.medium` cluster at the fixed price of $\$0.145$ per hour per node. We bid for a 10-node `c1.medium` cluster; the current market price is $\$0.02$ per hour per node. The density plots are generated from 5000 price traces. To make it easier to compare the three plots, we use the same (logarithmic) scale for all density maps, and use an arrow next to each map to mark the highest density in the corresponding plot.

Cost Estimation We begin by briefly describing how *Cumulon* estimates the completion time of a task. The two main contributors to task completion time are computation and network I/O. For each physical operator, we build a model to predict its computation time given machine type, number of slots per machine, operator parameter settings, input size, and input sparsity. *Cumulon* currently trains these models by benchmarking all machine types, though it would be helpful (and straightforward) to supplement training with performance traces of actual executions. We estimate the network I/O cost using an operator-specific model of the I/O volume, and a model for converting the I/O volume to time given the cluster sizes and configurations for primary and transient nodes. When modeling task completion time, we consider not only the expected time, but also variance—which is important in estimating job completion time, as explained below.

As explained in Section 2, *Cumulon* divides a job into multiple tasks and execute them in waves on all available slots. A naive estimate of job completion time would be the product of task completion time and the number of waves. However, this method works well only for a homogeneous cluster and when variance in task completion time is low. For example, Figure 5 shows the actual versus predicted costs of a matrix multiply job running on a homogeneous cluster (where cost equals the product of completion time, cluster size, and the per-node rate). Even in this simple setting, the naive estimates are inaccurate—the estimated cost fluctuates wildly as the cluster size varies, because the number of waves, an integral value, changes abruptly at specific cluster sizes. In contrast, the actual costs show a much smoother trend. The reason is that when task completion times vary, the scheduler assigns new tasks to slots that complete earlier, hence blurring the boundaries between waves and making the job completion times smoother. In predicting job completion times, *Cumulon* simulates the behavior of its scheduler for variable task completion times and potentially heterogeneous clusters. As Figure 5 exemplifies, the results are much more accurate than the naive estimates.

Optimization involving transient nodes requires costing *sync* jobs and the recovery phase. In both cases, *Cumulon* needs to estimate the number of tiles of a given matrix that are stored only at the transients nodes (and hence need to be copied or recovered). We derive this estimate by assuming that each tile is accessed by a node independently at a probability proportional to the processing power of the node, as more powerful nodes can complete more and/or bigger tasks and therefore access more tiles. The final estimate thus depends on the relative processing capacities of the primary and transient node clusters (a bigger transient cluster means fewer tiles will be stored at the primary nodes), as well as how many times an intermediate result is accessed in subsequent jobs (more accesses lead to more tiles cached at the primary nodes). To estimate recovery cost, *Cumulon* also needs to estimate the number of input matrix tiles required for recomputing a given subset of the output matrix tiles. To derive this estimate, we analyze input-output lineage for each operator, and assume that

the subset of tiles to be recovered is random.

Finally, *Cumulon* needs a model predicting the market price of transient nodes. Any stochastic model can be plugged in, provided that *Cumulon* can efficiently simulate the stochastic process. Currently, *Cumulon* uses a model with two components. First, we use non-parametric density estimation to approximate the conditional distribution of the future prices given the current and historical prices. To capture periodicity, we wrap the time dimension in a one-week cycle. Second, price spikes and their inter-arrival times are modeled as being conditionally independent given current and historical prices. We train the components using the historical prices of Amazon spot instances, and then combine the components to create a sample path for the stochastic price process. With this model, we can derive the distribution of the departure time of the transient nodes with a given bid price under the current market conditions. The bottom of Figure 4 plots an example departure time distribution. Using this distribution, we can then quantitatively compare the plans with and without *sync*, shown in the top of Figure 4, in terms of their expected costs and whether they meet the user-specified risk tolerance.

Optimization Algorithms As discussed in Section 1, *Cumulon* needs to explore a large, high-dimensional plan space. Although most *Cumulon* workloads are expensive enough to justify spending more time on optimization, we are still mindful of the optimization overhead, especially since we plan to make optimization online in the future. Thus, we use a number of techniques and assumptions to tame the complexity of optimization. First, *Cumulon* conducts a series of rule-based rewrites (e.g., linear algebra equivalences) on logical plans so that they lead to generally better physical plans with less computation and I/O or higher degree of parallelism. *Cumulon* then translates each logical plan into one or more alternative workflows of physical operators grouped into jobs; fewer jobs are usually preferred because of the overhead of initiating jobs during execution. For example, recall Figure 3: the merging of the second step of matrix multiply and the ensuing matrix add into a single job is carried out during this translation.

Given a job workflow, *Cumulon* still has to make many decisions to turn it into an executable plan. We solve the simpler optimization problem for plans without transient nodes by considering each possible machine type for the (primary node) cluster. Given the machine type, we bound the range of feasible cluster sizes and perform an exponential search for the optimal cluster size, exploiting the properties of the cost functions. For example, one such property is that as the cluster size increases, the execution time is generally non-increasing, while the cost is generally non-decreasing because of the growing parallelization overhead. As for the number of slots per machine, its choices are naturally limited on a given machine type, so we consider them exhaustively. Given the cluster configuration, we can then determine the optimal task size and other execution parameters for each physical operator independently. Finally, we consider cluster switching options. To bound the optimization time, we prioritize the search to explore plans with fewer switches first, and terminate the search when time runs out.

Starting with the baseline plan selected by the procedure above, *Cumulon* further optimizes the use of transient nodes. We consider each possible machine type for the transient nodes in turn. Beginning with the market price, we try successively higher bid prices in small increments ($\$0.01$ per hour). This simple search strategy is feasible because, fortunately, the bid price cannot go high under the risk tolerance constraint. Nonetheless, it is interesting to note that bidding *above* the regular, fixed price for the same machine type could be beneficial, because transient nodes are only charged at their market price, and a higher bid price will decrease the probability of their untimely departure. In general, it is difficult to make informed bids without *Cumulon*'s help. To see the effect of bid price more clearly, consider Figure 6, which compares the resulting cost distributions under three different bid prices. *Cumulon* computes these distributions using a collection of price traces obtained by repeatedly simulating from the stochastic market price model. With a low bid price, we can hope to get lucky and keep the transient nodes long enough to lower the cost; however, as shown by the high-density region on the left of Figure 6a, we are more likely to lose the transient nodes soon and end up paying more. With a high bid price, as shown in Figure 6c by the clear shift of density towards right, we are expected to have the transient nodes longer; however, we may not be able to improve the overall cost, because we also pay more for the tran-

sient nodes on average (as the market price can linger longer close to the higher bid price). The middleground (Figure 6b) between the two extremes may in fact be more attractive.

The decisions on the time and number of nodes to bid can be equally tricky for users. *Cumulon* considers bid times in small increments (10 minutes) until uncertainty becomes too high. *Cumulon* bounds the transient node cluster size by analyzing the parallelization overhead (as with the case of bounding the primary node cluster size), and by applying the risk tolerance constraint—even at low bid and market prices, a large number of transient nodes pose the risk of an untimely departure that may result in substantially higher cost than the baseline. Finally, *Cumulon* greedily adds *sync* jobs to the baseline plan: the *sync* job that gives the biggest improvement in expected cost will be picked, and the process repeats until no more improvement can be made.

In summary, *Cumulon* features a cost-based, uncertainty-aware optimizer that needs to deal with a large plan space. It takes a sampling-based approach towards handling uncertainty, and it keeps the optimization time under control using a combination of heuristics, practical constraints, and search techniques such as branch-and-bound. For example, on a regular laptop with an Intel i7-2600 CPU and 8G of memory, for RSVD-1 with 10 jobs, *Cumulon* is able to solve the full-fledged optimization problem—which involves finding optimal plans with bid times up to about one day into future, computing their expected costs, and quantifying the associated uncertainty in order to test whether they meet the given risk tolerance—under just one minute [11].

4 Related Work

Many projects are aimed at supporting statistical computing in the cloud. Some expose users to lower-level parallel programming abstractions, e.g., *RHIPE* [8], *Ricardo* [5], Revolution Analytics' *R/Hadoop*, and *MLI* [21]. Some provide higher-level libraries, e.g., Apache *Mahout* and *HAMA* [19]. Such libraries significantly simplifies development if they happen to offer the exact same algorithms needed by the task at hand, but they provide little help with the development of new algorithms beyond reusing implementations of low-level primitives. Furthermore, most of these libraries still have performance knobs that are difficult to tune for users. In contrast, by automatically parallelizing and optimizing programs expressed in terms of matrix primitives, *Cumulon* can simplify the development and deployment of even brand new algorithms.

Besides *Cumulon*, a number of other projects also support automatic optimization of statistical computing workloads. For example, *MADlib* [4, 9] leverages optimization techniques for parallel databases. *SystemML* [7, 2] takes an approach similar to ours, but only considers optimization of execution, while *Cumulon* additionally considers hardware provisioning and configuration setting as well as bidding strategies for transient nodes. The root of this difference lies in *Cumulon*'s focus on the users' perspective: while other systems optimize execution time on a given cluster, *Cumulon* goes a step further and asks what cluster would be most cost-effective (and how to bid for it) in the first place. Although automatic resource provisioning has been studied recently for general MapReduce systems [13, 23, 29], our focus on declaratively specified matrix-based programs leads to different challenges and opportunities in joint optimization of provisioning and execution that better exploits program semantics.

There has also been a lot of related work from the HPC community on automatic optimization of matrix-based programs, though none has gone as far as automatically making provisioning and configuration decisions. Most closely related to *Cumulon* is the work on "semantic" optimization of MATLAB programs (i.e., at the level of linear algebra operators) [17]. At one point, MATLAB provided a chain-matrix-multiply function that reordered multiplies using associativity. Generally speaking, however, platforms such as MATLAB and R do not automatically perform high-level linear algebra rewrites.[4] Beyond optimization at the level of linear algebra

[4]One reason why these platforms do not support linear algebra rewrite is the dynamic, interpreted nature of these languages. Another reason is the concern for reproducibility: for example, although in theory matrix multiplies are associative, in practice different orders can lead to different results due to errors inherent in the machine representation of numerical values. In *Cumulon*, we take the view that such concerns should not preclude automatic optimization by default—only in cases where such errors matter should users disable

expressions, there is also a long line of work from the compiler community on optimizing array code using loop analysis and transformations. Indeed, we have applied compiler techniques in a precursor project to *Cumulon* called *RIOT* [27, 28] to improve the I/O efficiency of matrix-based programs. Although *Cumulon* currently does not optimize at the loop level, how to exploit both styles of optimization in *Cumulon* remains an interesting possibility. We refer interested readers to [27, 28] for discussion of related work in this area.

Support for auction-based markets of cloud computing resources has also attracted much attention. A number of systems support using transient nodes for MapReduce: e.g., Amazon's own *Elastic MapReduce*, Qubole's auto-scaling clusters, and [15, 3]. They focus on storage and execution support; a more in-depth comparison with the Cumulon's dual-HDFS approach can be found in [11]. They do not help users choose bidding strategies or optimize their programs. Bidding strategies have been studied in [22, 24, 1, 25]; some of the approaches also models market price stochastically. However, unlike *Cumulon*, they assume black-box jobs, and hence do not have the knowledge of program semantics to optimize bidding strategies and execution plans jointly. For example, they employ generic checkpointing and/or migration techniques, but cannot extend plans with *sync* jobs intelligently as *Cumulon* does. Finally, support for auction-based markets has also been studied from the perspective of service providers [16, 20], which complements *Cumulon*'s focus on the perspective of users.

5 Conclusion

The increasing commoditization of computing, aided by the growing popularity of public clouds, holds the promise to make big-data analytics accessible to more users than ever before. However, in contrast to the relative ease of obtaining hardware resources, most users still find it difficult to use these resources effectively. In this paper, we have presented *Cumulon*, an ongoing project at Duke University aimed at simplifying—from users' perspective—both development and deployment of large-scale, matrix-based data analysis on public clouds. *Cumulon* is about providing the appropriate abstractions to users: it applies the philosophy of database systems—"*specify what you want, not how to do it*"—to the world of matrices, linear algebra, and cloud. As discussed in this paper, achieving this vision involves many challenges. *Cumulon* builds on existing cloud programming platforms, but carefully avoids the limitations of their models and dependency on specific platforms using a simple yet flexible storage and execution framework, which also allows database- and HPC-style processing to be incorporated for higher efficiency. *Cumulon* automatically makes cost-based decisions on a large number of issues, from the setting of execution parameters to the choice of clusters and bidding strategies. *Cumulon* also fully embraces uncertainty, not only for better performance modeling, but also in formulating its optimization problem to capture users' risk tolerance. The combination of these design choices and techniques enables *Cumulon* to provide an end-to-end, user-facing solution for running matrix-based workloads in the cloud.

Future Work As discussed in Section 3, we are actively investigating online optimization in *Cumulon*. While *Cumulon* alerts the user upon detecting significant deviations from the predicted execution progress or market price, it currently relies on the user to take further actions. We would like to make *Cumulon* adapt to changing markets, recover from mispredictions, and deal with unexpected events intelligently, with less user supervision.

Our work on *Cumulon* to date has mostly focused on the *Cumulon* backend; we are still working on devising friendly interfaces for programming, performance monitoring, and interactive optimization. Instead of a full-fledged language, we envision a high-level language targeting just the domain of linear algebra, similar in syntax to MATLAB and R. Users write matrix-based, compute-intensive parts of their analysis in this language. These parts are embedded in a program written in a host language (such as R) and preprocessed by *Cumulon*. Users specify their time, money, and risk tolerance requirements (as well as other pertinent information such as authentication information) in a deployment descriptor that go along with the program. We also plan to develop

rewrites explicitly.

a graphical user interface for *Cumulon*, to better present output from the optimizer and the performance monitor, and to make input more user-friendly.

Accurate performance prediction for arbitrary matrix computation is challenging. Many analysis techniques are iterative, and their convergence depends on factors ranging from data characteristics (e.g., condition numbers of matrices) to starting conditions (e.g., warm vs. cold). So far, *Cumulon* has taken a pragmatic approach—it focuses on optimizing and costing a single or fixed number of iterations. User need to supply the desired number of iterations, or reason with "rates" of progress and cost of iterative programs. Better techniques for predicting convergence will be useful, but it remains unclear whether we can bound the prediction uncertainty to low enough levels to meet users' risk tolerance requirements. Making the optimizer online may be our best option.

Cumulon currently only exploits intra-job parallelism. However, there are also opportunities for inter-job parallelism in statistical analysis workloads [14]. It would be nice to support both forms of parallelism. *SystemML* has made promising progress in this direction [2], but we will need to extend the optimization techniques to consider cluster provisioning and bidding strategies in this setting.

To keep up with rapid advances in big-data analysis, we must make *Cumulon* an extensible and evolvable platform. Developers should be able to contribute new computational primitives to *Cumulon* in a way that extends not only its functionality but also its optimizability. While implementing a new operator may be easy, telling *Cumulon* how to optimize the use of this new operator is much harder. *Cumulon* relies on high-quality cost models that come with assessment of prediction uncertainty, but we cannot always assume such models to be available from the start; instead, we need techniques for automatically building reasonable "starter" models as well as identifying and improving poor models, in order to provide meaningful risk assessment for users. In general, supporting extensibility in optimizability—from cost estimation to search algorithms—poses many challenges that have until now been relatively less studied.

Acknowledgments This work has been supported by NSF grants 0916027, 0917062, 0964560, 1320357, a 2010 HP Labs Innovation Research Award, and grants from Amazon Web Services.

References

[1] A. Andrzejak, D. Kondo, and S. Yi. Decision model for cloud computing under SLA constraints. In *Proceedings of the 2010 IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 257–266, Miami, Florida, USA, Aug. 2010.

[2] M. Boehm, S. Tatikonda, B. Reinwald, P. Sen, Y. Tian, D. Burdick, and S. Vaithyanathan. Hybrid parallelization strategies for large-scale machine learning in SystemML. *Proceedings of the VLDB Endowment*, 7(7):553–564, 2014.

[3] N. Chohan, C. Castillo, M. Spreitzer, M. Steinder, A. N. Tantawi, and C. Krintz. See spot run: Using spot instances for MapReduce workflows. In *Proceedings of the 2010 Workshop on Hot Topics on Cloud Computing*, Boston, Massachusetts, USA, June 2010.

[4] J. Cohen, B. Dolan, M. Dunlap, J. M. Hellerstein, and C. Welton. MAD skills: New analysis practices for big data. In *Proceedings of the 2009 International Conference on Very Large Data Bases*, pages 1481–1492, Lyon, France, Aug. 2009.

[5] S. Das, Y. Sismanis, K. S. Beyer, R. Gemulla2, P. J. Haas, and J. McPherson. Ricardo: Integrating R and Hadoop. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, Indianapolis, Indiana, USA, June 2010.

[6] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of the 2004 USENIX Symposium on Operating Systems Design and Implementation*, pages 137–150, San Francisco, California, USA, Dec. 2004.

[7] A. Ghoting, R. Krishnamurthy, E. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian, and S. Vaithyanathan. SystemML: Declarative machine learning on MapReduce. In *Proceedings of the 2011 International Conference on Data Engineering*, Hannover, Germany, Apr. 2011.

[8] S. Guha. *Computing Environment for the Statistical Analysis of Large and Complex Data*. PhD thesis, Purdue University, 2010.

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

[10] B. Huang, S. Babu, and J. Yang. Cumulon: Optimizing statistical data analysis in the cloud. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, New York City, New York, USA, June 2013.

[11] B. Huang, N. W. D. Jarrett, S. Babu, S. Mukherjee, and J. Yang. Leveraging spot instances for cloud-based statistical data analysis. Technical report, Duke University, July 2014.

[12] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2007 EuroSys Conference*, pages 59–72, Lisbon, Portugal, Mar. 2007.

[13] K. Kambatla, A. Pathak, and H. Pucha. Towards optimizing hadoop provisioning for the cloud. In *Proceedings of the 2009 Workshop on Hot Topics on Cloud Computing*, Boston, Massachusetts, USA, June 2009.

[14] J. Li, X. Ma, S. B. Yoginath, G. Kora, and N. F. Samatova. Transparent runtime parallelization of the R scripting language. *Journal of Parallel and Distributed Computing*, 71(2):157–168, 2011.

[15] H. Liu. Cutting MapReduce cost with spot market. In *Proceedings of the 2011 Workshop on Hot Topics on Cloud Computing*, Portland, Oregon, USA, June 2011.

[16] M. Mazzucco and M. Dumas. Achieving performance and availability guarantees with spot instances. In *Proceedings of the 2011 IEEE International Conference on High Performance Computing and Communication*, pages 296–303, Banff, Alberta, Canada, Sept. 2011.

[17] V. Menon and K. Pingali. High-level semantic optimization of numerical codes. In *Proceedings of the 1999 ACM/IEEE Supercomputing Conference*, pages 434–443, Rhodes, Greece, June 1999.

[18] V. Rokhlin, A. Szlam, and M. Tygert. A randomized algorithm for principal component analysis. *SIAM Journal on Matrix Analysis and Applications*, 31(3):1100–1124, Aug. 2009.

[19] S. Seo, E. J. Yoon, J.-H. Kim, S. Jin, J.-S. Kim, and S. Maeng. HAMA: An efficient matrix computation with the MapReduce framework. In *Proceedings of the 2010 IEEE International Conference on Cloud Computing Technology and Science*, pages 721–726, Indianapolis, Indiana, USA, Nov. 2010.

[20] Y. Song, M. Zafer, and K.-W. Lee. Optimal bidding in spot instance market. In *Proceedings of the 2012 IEEE International Conference on Computer Communications*, pages 190–198, Orlando, Florida, USA, Mar. 2012.

[21] E. R. Sparks, A. Talwalkar, V. Smith, J. Kottalam, X. Pan, J. E. Gonzalez, M. J. Franklin, M. I. Jordan, and T. Kraska. MLI: An API for distributed machine learning. In *Proceedings of the 2013 International Conference on Data Mining*, pages 1187–1192, Dallas, Texas, USA, Dec. 2013.

[22] S. Tang, J. Yuan, and X.-Y. Li. Towards optimal bidding strategy for Amazon EC2 cloud spot instance. In *Proceedings of the 2012 IEEE International Conference on Cloud Computing*, number 91–98, Honolulu, Hawaii, USA, June 2012.

[23] A. Verma, L. Cherkasova, and R. H. Campbell. ARIA: automatic resource inference and allocation for MapReduce environments. In *Proceedings of the 2011 International Conference on Autonomic Computing*, Karlsruhe, Germany, June 2011.

[24] W. Voorsluys and R. Buyya. Reliable provisioning of spot instances for compute-intensive applications. In *Proceedings of the 2012 IEEE International Conference on Advanced Information Networking and Applications*, pages 542–549, Fukuoka, Japan, Mar. 2012.

[25] S. Yi, A. Andrzejak, and D. Kondo. Monetary cost-aware checkpointing and migration on Amazon cloud spot instances. *IEEE Transactions on Services Computing*, 5(4):512–524, 2012.

[26] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2010 Workshop on Hot Topics on Cloud Computing*, Boston, Massachusetts, USA, June 2010.

[27] Y. Zhang, H. Herodotou, and J. Yang. RIOT: I/O-efficient numerical computing without SQL. In *Proceedings of the 2009 Conference on Innovative Data Systems Research*, Asilomar, California, USA, Jan. 2009.

[28] Y. Zhang and J. Yang. Optimizing I/O for big array analytics. *Proceedings of the VLDB Endowment*, 5(8):764–775, June 2012.

[29] Z. Zhang, L. Cherkasova, A. Verma, and B. T. Loo. Meeting service level objectives of Pig programs. In *Proceedings of the 2012 International Workshop on Cloud Computing Platforms*, pages 8:1–8:6, Bern, Switzerland, Apr. 2012.

Data Engineering

It's FREE to join!

TCDE

tab.computer.org/tcde/

The Technical Committee on Data Engineering (TCDE) of the IEEE Computer Society is concerned with the role of data in the design, development, management and utilization of information systems.

- Data Management Systems and Modern Hardware/Software Platforms
- Data Models, Data Integration, Semantics and Data Quality
- Spatial, Temporal, Graph, Scientific, Statistical and Multimedia Databases
- Data Mining, Data Warehousing, and OLAP
- Big Data, Streams and Clouds
- Information Management, Distribution, Mobility, and the WWW
- Data Security, Privacy and Trust
- Performance, Experiments, and Analysis of Data Systems

The TCDE sponsors the International Conference on Data Engineering (ICDE). It publishes a quarterly newsletter, the Data Engineering Bulletin. If you are a member of the IEEE Computer Society, you may join the TCDE and receive copies of the Data Engineering Bulletin without cost. There are approximately 1000 members of the TCDE.

Join TCDE via Online or Fax

ONLINE: Follow the instructions on this page:

www.computer.org/portal/web/tandc/joinatc

FAX: Complete your details and fax this form to **+61-7-3365 3248**

Name _____

IEEE Member # _____

Mailing Address _____

_____

Country _____

Email _____

Phone _____

TCDE Mailing List

TCDE will occasionally email announcements, and other opportunities available for members. This mailing list will be used only for this purpose.

Membership Questions?

Xiaofang Zhou
School of Information Technology and Electrical Engineering
The University of Queensland
Brisbane, QLD 4072, Australia
zxf@uq.edu.au

TCDE Chair

Kyu-Young Whang
KAIST
371-1 Koo-Sung Dong, Yoo-Sung Ku
Daejeon 305-701, Korea
kywhang@cs.kaist.ac.kr

IEEE Computer Society
1730 Massachusetts Ave, NW
Washington, D.C. 20036-1903

Non-profit Org.
U.S. Postage
PAID
Silver Spring, MD
Permit 1398