DOI:10.1145/2428556.2428570

Article development led by acmqueue
queue.acm.org

Racing to unleash the full potential of big data with the latest statistical and machine-learning techniques.

BY ARUN KUMAR, FENG NIU, AND CHRISTOPHER RÉ

Hazy: Making It Easier to Build and Maintain Big-Data Analytics

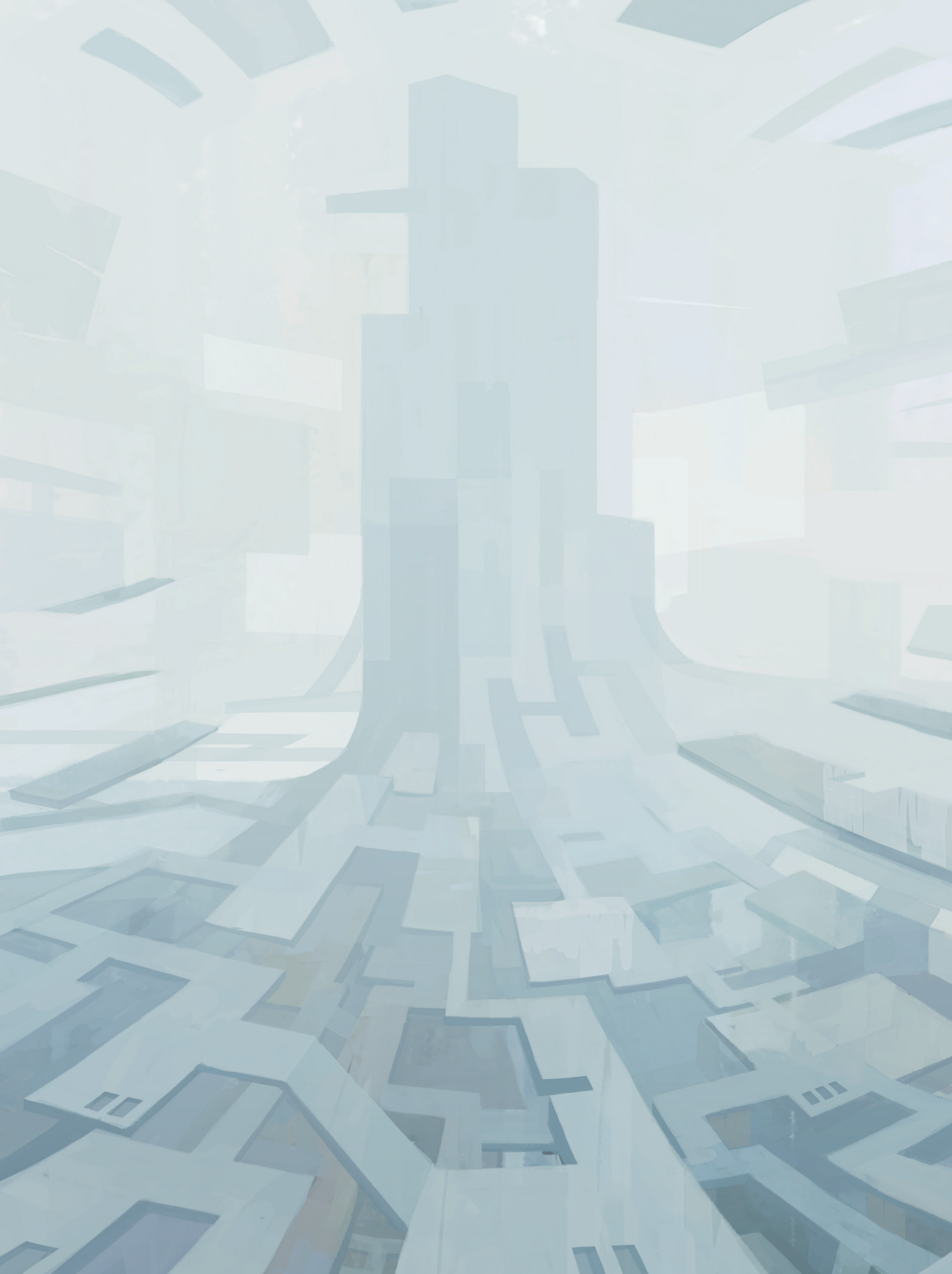
THE RISE OF big data presents both big opportunities and big challenges to domains ranging from enterprises to sciences. The opportunities include better-informed business decisions, more efficient supply-chain management and resource allocation, more effective targeting of products and advertisements, better ways to "organize the world's information," and faster turnaround of scientific discoveries, among others.

The challenges are also tremendous. For one, more data comes in diverse forms: such as text, audio, video, OCR, and sensor data. While existing data management systems predominantly assume that data has rigid, precise semantics, increasingly more data (albeit valuable) contains imprecision or inconsistency. For another, the proliferation of ever-evolving algorithms to gain insights from data (in the name of machine learning, data mining, statistical analysis, and so on) can often be daunting to a developer with a particular dataset and specific goals: the developer not only has to keep up with the state of the art, but also must expend significant development effort in experimenting with different algorithms.

Many state-of-the-art approaches to both of these challenges are largely statistical and combine rich databases with software driven by statistical analysis and machine learning. Examples include Google's Knowledge Graph, Apple's Siri, IBM's "Jeopardy!"-winning Watson system, and the recommendation systems of Amazon and Netflix. The success of these big-data analytics-driven systems, also known as *trained systems*, has captured the public imagination, and there is excitement in bringing such capabilities to other verticals such as enterprises, health care, sciences, and government. The complexity of such systems, however, means that building them is very challenging, even for Ph.D.-level computer scientists. If such systems are to have truly broad impact, building and maintaining them needs to become substantially easier, so that they can be turned into commodities that can be easily applied to different domains. Most of the research emphasis so far has been on individual algorithms for specific machine-learning tasks.

In contrast, the Hazy project (http://hazy.cs.wisc.edu) takes a systems approach with the hypothesis: The next breakthrough in data analysis may not be in individual algorithms, but in the ability to rapidly combine, deploy,

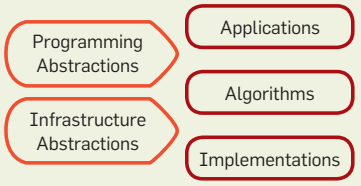
ILLUSTRATION BY MARK BOARDMAN



and maintain existing algorithms. Toward that goal, Hazy's research has focused on identifying and validating two broad categories of "common patterns" (also known as *abstractions*) in building trained systems (see Figure 1): programming abstractions and infrastructure abstractions. Identifying, optimizing, and supporting such abstractions as primitives could make trained systems substantially easier to build. This can bring us a step closer to unleashing the full potential of big-data analytics in various domains.

Programming abstractions. To ensure that a trained-system platform is accessible to many developers, the programming interface must be small and composable to enhance productivity and enable developers to try many algorithms; the ability to integrate diverse data resources and formats requires the data model of the programming interface to be versatile. A combination of the relational data model and a probabilistic rule-based language such as Markov logic satisfies these criteria. Using this combination, we have developed several knowledge-based construction systems (namely, DeepDive, GeoDeepDive, and AncientText). Furthermore, our (open source) software stack has been downloaded thousands of times and used by different communities such as natural language processing, chemistry, and biostatistics.

Figure 1. Hazy's programming abstractions and infrastructure abstractions.



Infrastructure abstractions. To build a trained-system platform that can accommodate many different algorithms and that scales to large volumes of data, it is crucial to find the invariants in applying individual algorithms, to have a clean interface between algorithms and systems, and to have a scalable data-management and memory-management subsystem. Using these principles, we developed a prototype system called Bismarck,[10] which leverages the observation that many statistical-analysis algorithms behave as a user-defined aggregate in an RDBMS. The Bismarck approach to data analysis is resonated by commercial systems providers such as Oracle and EMC Greenplum. In addition, such infrastructure-level abstractions allow us to explore generic techniques for improving the scalability and efficiency of many algorithms.

Example Application: GeoDeepDive

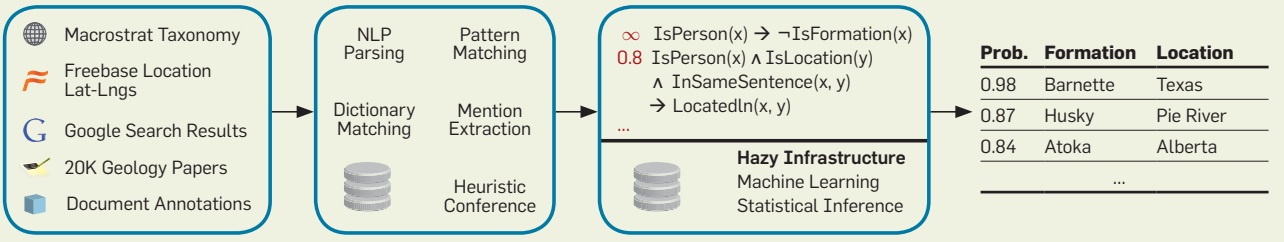
An application called GeoDeepDive (http://hazy.cs.wisc.edu/geodeepdive) illustrates the Hazy approach to building trained systems. GeoDeepDive is a demo project involving collaboration with geology researchers to perform deep linguistic and statistical analysis over a corpus of tens of thousands of journal papers in geology. The goal is to extract useful information from this corpus and organize it in a way that facilitates geologists' research. The current version of GeoDeepDive extracts mentions of rock formations, tries to assign various types of attributes to these formation mentions (for example, location, time interval, carbon measurements), and then organizes the extractions and documents in spatial and temporal dimensions for geoscientists. Figure 2 shows a high-level overview of how Hazy built GeoDeepDive.

Using the Hazy approach, the GeoDeepDive's development pipeline consists of the following steps:

1. The developer assembles data resources that are potentially useful for GeoDeepDive.
2. The developer composes feature-extraction functions that convert the data resources into relational signals.
3. The developer specifies correlations and constraints over the relational signals in the form of probabilistic rules; Hazy's infrastructure performs scalable statistical learning and inference automatically.
4. Hazy outputs probabilistic predictions for GeoDeepDive.

Input data sources. Hazy embraces all data sources that can be useful for an application. GeoDeepDive uses the Macrostrat taxonomy (http://macrostrat.org/) because it provides the set of entities of interest, as well as domain-specific constraints (for example, a formation can be associated only with certain time intervals). Google search results are used to map location mentions to their canonical names and then to latitude-longitude (lat-lng) coordinates using Freebase (http://freebase.com). These coordinates can be used to perform geographical matching against the formations' canonical locations (lat-lng polygons in Macrostrat). There are also (manual)

Figure 2. An overview of GeoDeepDive's development pipeline.



(a) Input Data Sources (b) Feature Extraction (c) Statistical Processing (d) Output Probabilistic Predictions

document annotations of textual mentions of formation measurements that serve as training data.

Feature extraction. The input data sources may not have the desired format or semantics to be used directly as *signals* (or *features*) for statistical inference or learning. The feature-extraction step performs such conversions. The developer specifies the schema of all relations, provides individual extractors, and then specifies how these extractors are composed together. For example, we (the developers) perform NLP parsing on the input corpus to produce per-sentence structured data such as part-of-speech tags and dependency paths. We then use the Macrostrat taxonomy and heuristics to extract candidate entity mentions (of formations, measures, among others), as well as possible co-reference relationships between the mentions.

Statistical processing. The signals produced by feature extraction may contain imprecision or inconsistency. To make coherent predictions, the developer provides constraints and (probabilistic) correlations over the signals. Hazy uses the Markov logic language; a Markov logic program consists of a set of weighted logical rules that represent high-level constraints or correlations. The developer may also specify available training data, which Hazy would use to learn rule weights. The programming interface isolates the internals of Hazy's statistical processing from the developer. It is in Hazy's infrastructure where the developer can plug in various algorithms.

Output probabilistic predictions. The output from Hazy's statistical processing infrastructure consists of probabilistic predictions on relations of interest (for example, `LocatedIn` in Figure 2). In general, Hazy prefers algorithms with theoretical guarantees (for example, Gibbs sampling). Such algorithms ensure the output predictions are well calibrated (for example, if all predictions with probability 0.7 are examined, then close to 70% of these predictions are correct). These predictions can then be fed into the front end of GeoDeepDive (Figure 3).

In addition to GeoDeepDive, we have deployed the Hazy approach in several other projects in a similar manner—for example, DeepDive (http://hazy.cs.wisc.edu/deepdive), which enhances Wikipedia with facts extracted from the Web[16] (see Figure 4).

Programming Abstractions

Programming abstractions decouple the developer's application-specific (logical and statistical) modeling from the (statistical inference or learning) algorithms to be used for an application at execution time. The purpose of such abstractions is to ensure: an application developer can try many different algorithms for the same dataset and/or domain knowledge or heuristics without additional development effort; and when the efficiency or quality of one algorithm improves, all applications using this algorithm experience automatic improvement. A combination of the relational data model and a probabilistic logic-based programming language has proved effective for meeting these two criteria.

Relational data model. As seen in the GeoDeepDive example, Hazy's approach to statistical data analysis uses the relational data model as the basis for the programming abstractions. Apart from being well studied,

Figure 3. Screen shot showing probabilistic predictions in GeoDeepDive.

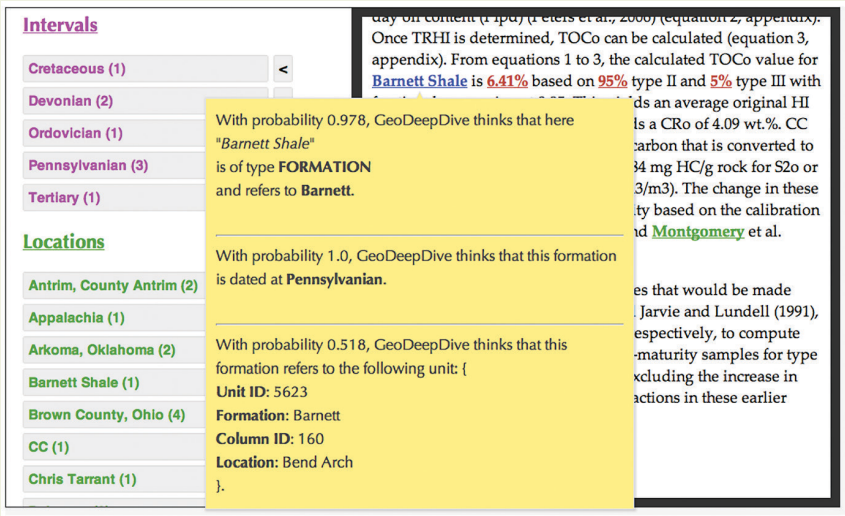


Figure 4. Sample relations about Barack Obama, Elon Musk, and Microsoft extracted by DeepDive.

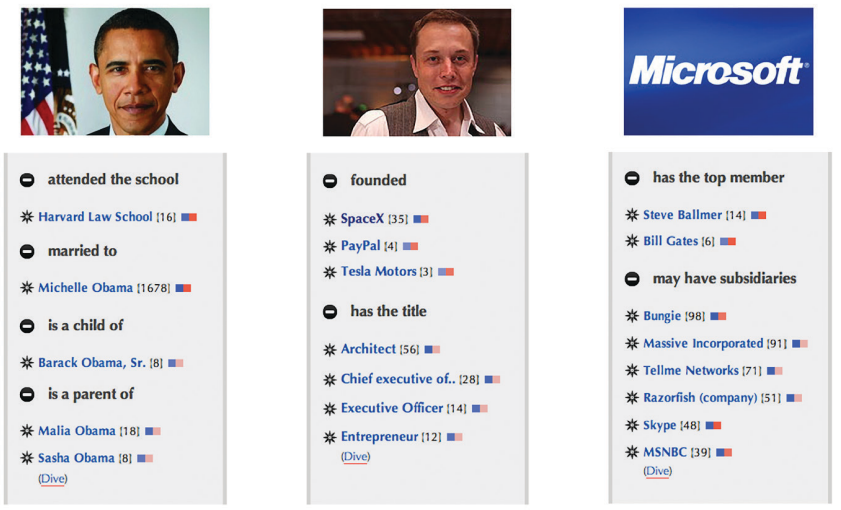


Figure 5. Comparison of in-memory grounding of Markov logic with in-RDBMS grounding.

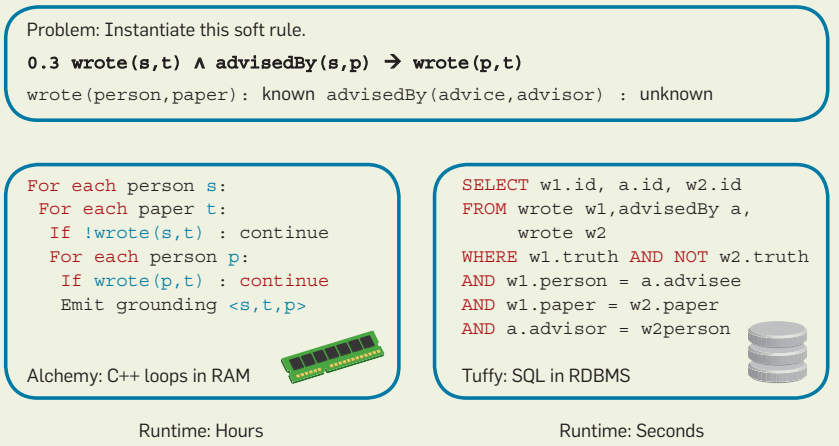
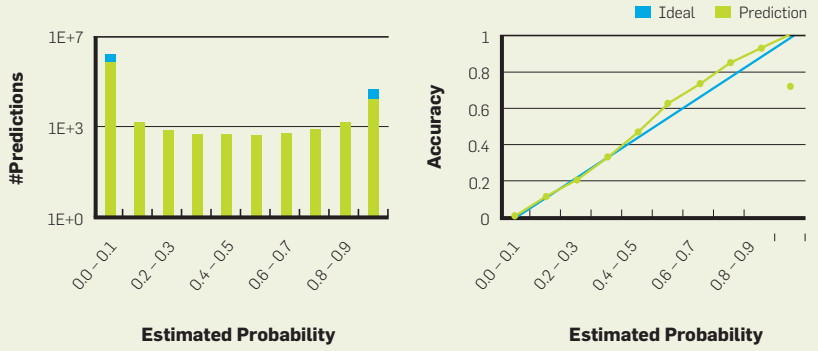


Figure 6. Macro-debugging: Example probability calibration graphs for a text-chunking task.



the relational model also underlies a large class of important statistical and machine-learning methods that use relational-style feature vectors. As an important consequence, this choice automatically provides the advantages of a mature data platform such as an RDBMS. For example, using an RDBMS to manage the data in a Hazy pipeline (as in Figure 2), a developer can easily perform data loading from and to other systems. Moreover, as RDBMS technologies continue to mature and evolve, the same Hazy pipeline would continue to gain in scalability and performance automatically.

Probabilistic logic programming. The intuitiveness, flexibility, and growing popularity of Markov logic[18] led to its adoption as a central programming language in Hazy. Researchers have applied it to a wide range of applications. In Markov logic, a developer can write first-order logic rules with weights (which intuitively model one's confidence in a rule); this allows the developer to capture rules that are likely, but not certain, to be correct. A Markov logic program (also known as Markov logic network, or simply MLN) specifies what data (evidence) is available, what predictions to make, and what constraints and correlations exist. The process of computing predictions given an MLN is called *inference*. Sometimes an MLN may be missing weights, and a developer can provide training data from which Hazy can *learn* rule weights.

Semantically, an MLN represents a probabilistic graphical model (conceptually via rule instantiation) that in turn represents a probabilistic distribution over all possible configurations of the relations in an application. Thus, a key advantage of Markov logic is its succinctness. On the other hand, such succinctness also introduces a technical challenge of efficiently instantiating (or grounding) the first-order rules. Our crucial observation is that grounding fundamentally does relation-style joins. We built an RDBMS-based MLN interface engine, Tuffy, which leverages time-tested RDBMS infrastructure for joins to achieve high performance at scale.[14] As it turned out, Tuffy was much faster and more scalable than the state-of-the-art MLN inference engine at the time (see Figure 5). The comparison is between Alchemy's in-memory grounding (that is, rule instantiation) of Markov logic and Tuffy's in-RDBMS grounding. Both code snippets are automatically generated by the corresponding systems for the afore-mentioned MLN rule. The use of an RDBMS makes Tuffy scalable and orders-of-magnitude faster than Alchemy, since Tuffy leverages mature RDBMS infrastructure for joins.

Markov logic is a flexible language, allowing the developer to easily represent common statistical models such as logistic regression and conditional random fields; furthermore, the developer can build more sophisticated statistical models by combining multiple "primitive" models or adding additional correlations or constraints.[18] Internally, the Hazy infrastructure is able to recognize certain "primitive" models in an MLN and select inference or learning algorithms accordingly. Still, some useful statistical modeling elements are not easily represented in Markov logic (for example, correlations involving continuous random variables or aggregations). To support these more sophisticated modeling functionalities, we are extending Hazy's programming interface to support general factor-graph construction. We are also working on extending the framework to support user-defined functions for richer application-specific logic.

Debugging. From our experience with GeoDeepDive and DeepDive, we have found debugging to be a key task in developing a trained system. Debugging is the process of performing corrections or fine-tuning to the components of an application (say, a feature extractor or an MLN program). It is error-prone and often tedious. To facilitate the debugging process, we con-

sider it to be an integral component in programming. Here are two types of debugging that are applicable for statistical data processing in general:

▸ *Macro-debugging with calibration graphs.* For probabilistic predictions to make sense, they must be well calibrated. For example, if a system outputs a prediction with probability 0.7, we want the accuracy of this prediction to be 70%. A calibration graph characterizes how prediction accuracy changes with respect to prediction probabilities. In Figure 6 the *x*-axis is the probability of predictions estimated, and the *y*-axis on the left is the number of predictions made by the system (accuracy of predictions). Intuitively, if the system outputs a prediction with probability 0.7, we want the accuracy of this prediction to be 70%. These results are for a skip-chain CRF (conditional random field) model used on the CoNLL-2000 (Conference on Computational Natural Language Learning) text-chunking task, which contains a training set used for training the model, and a testing set used for evaluation. Gibbs sampling runs until convergence (decided by the Wald test) and provides inference results on the testing set. Such assessment serves as a sanity check of the whole system; if we discover that a system is not well calibrated, we can look into the training-data acquisition process and check possible overfitting problems.

▸ *Micro-debugging with error analysis.* To refine or add more probabilistic rules, an effective approach is to analyze errors in the results that our system produces: a developer annotates each prediction as either correct or incorrect, classifies errors into different groups, and then addresses the error groups accordingly. The challenge is that to annotate one prediction, we may need to consult many related relations. The saving grace is that, with a modeling language such as Markov logic, it is possible to trace backward from each prediction to the originating signals (and the rules in between). We are trying to design and implement a debugging IDE (integrated development environment) to support such explanations using provenance information.

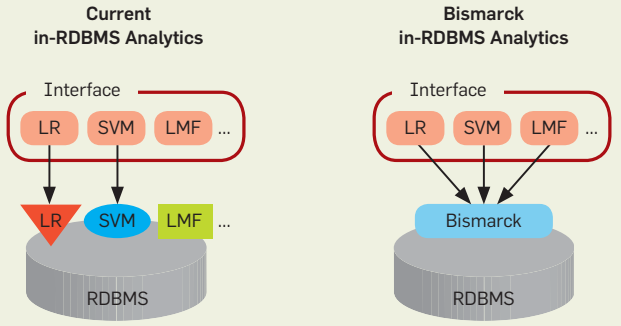
Bismarck: A Unified Architecture for in-RDBMS Analytics

The Bismarck project[10] is the first step in devising common infrastructure abstractions. Infrastructure abstractions are what decouple algorithms from implementation details such as data management, memory management, and task scheduling. Having a clean infrastructure abstraction ensures a system builder does not have to reinvent or reengineer the wheel when adding a new algorithm into the system; and when one component of the infrastructure is improved (for example, better memory management), all algorithms benefit automatically. Furthermore, a clean infrastructure abstraction provides clear angles to investigate generic techniques for improving broad classes of algorithms.

Motivation. The Bismarck project was motivated by the trend of bringing sophisticated data analytics into enterprise applications that depend on an RDBMS. From our conversations with engineers from Oracle and EMC Greenplum,[13] we learned that the overhead of building each new analytics technique from scratch—implementing a new solver with new memory requirements, data access methods, and others—was a major bottleneck in practice. Bismarck aims to simplify such systems with a *unified infrastructural abstraction* to handle many techniques.

Convex programming: A unifying mathematical abstraction. We begin with an important observation from the math programming literature: many analytics techniques can be framed as *convex programming problems*.[8,12] A convex program is an optimization problem where the objective function is convex (bowl-shaped). Examples include logistic regression, SVMs (support-vector machines), and conditional random fields. Not all problems are convex (for example, Apriori[1] and some graph-mining algorithms), but a large class of problems are convex (or convex relaxations), as illustrated in Figure 7a. In contrast to existing in-RDBMS analytics tools that have separate code paths for dif-

Figure 7. (a) Bismarck in an RDBMS; (b) An incomplete list of tasks and techniques.



(a)

Analytics Task or Technique
Logistic Regression (LR)
Support Vector Machine (SVM)
Low-Rank Matrix Factorization (LMF)
Conditional Random Field (CRF)
Least-squares, Lasso, and Ridge Regression
Graph Max-Cut Problems
Kalman Filters
Portfolio Optimization

(b)

ferent analytics techniques, Bismarck provides a single framework to implement them, while possibly retaining the same interfaces. Figure 7b shows an incomplete list of tasks and techniques that can be handled by Bismarck using convex programming (and convex relaxations).

This observation is very important in data-analysis theory, since researchers are able to unify their algorithmic and theoretical studies of such problems. Convex problems are attractive since local solutions are always globally optimal, and there are many well-studied algorithms that can solve them. Because convex programming allows the problem definition to be decoupled from the way it is solved or implemented, it is a natural starting point for a unified analytics architecture.

Many analytics techniques have convex objective functions that are also linearly separable: formally, the problem is to find a vector $w \in \mathbb{R}^d$ (the *model*) for some $d \geq 1$ that minimizes the following objective:

$$\min_{w \in \mathbb{R}^d} F(w) = \sum_{i=1}^{N} f(w, z_i) + P(w)$$

The objective function $F(w)$ is a sum of terms $f(w, z_i)$ for $i = 1, \ldots, N$ where each z is a (training) data point. In Bismarck, the z_i is represented by database tuples—for example, (paper, area) for paper classification. We abbreviate $f(w, z_i) = f_i(w)$. For example, in SVM classification, $f_i(w)$ is the hinge loss of the model w on the ith data point, and $P(w)$ enforces the smoothness of the classifier (preventing overfitting). We can generalize this to include constraints via proximal point methods. One can also generalize to both matrix valued w and nondifferentiable functions.[19]

Gradient methods and incremental gradient descent. There are many well-studied algorithms to solve convex programs, and the most popular are the *gradient methods*. A *gradient*, formally denoted by ∇F, is the generalization of the derivative of a function. Essentially, it gives the slope of the curve at a point, as shown in Figure 8a. The gradient is linear, which means ∇F can be computed as the sum of the N individual gradients $\nabla f(w, z_i)$.

Gradient methods are iterative algorithms that solve convex programs (1). They start at some initial value for w and then compute the gradient (and/or related quantities) and use it to take a step to the next value of w, until the method converges to an optimum. Popular gradient methods include Conjugate Gradient, Newton Method, and BFGS.[8] They all scan the full dataset at each iteration to compute the full gradient ∇F for a single step. This could make them inefficient for big data. Our goal is to choose a gradient method whose data-access properties are amenable to an efficient in-RDBMS implementation. A classical algorithm called IGD (*Incremental Gradient Descent*) fits the bill. IGD approximates the full gradient ∇F using only one term at a time. Formally, assuming $P = 0$ for simplicity, IGD updates the current value at iteration k, $w^{(k)}$ using a rule such as:

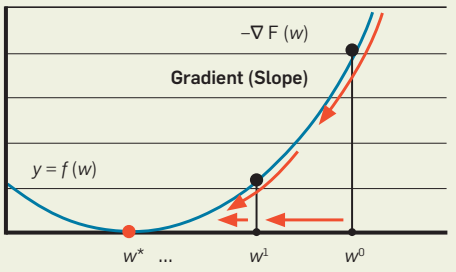
$$w^{(k+1)} = w^{(k)} - \alpha_k \nabla f(w^{(k)}, z_j)$$

Here, $\alpha k \geq 0$ is a parameter called *stepsize*, while z_i is one data point. In a database, each z_i corresponds to one tuple, which brings us to our central observation: IGD has a tuple-at-a-time data-access pattern that is essentially identical to a SQL aggregate such as AVG. Essentially, IGD looks at the data tuple one at a time and performs a (noncommutative) "aggregation." IGD is also a fast algorithm, with a runtime that is linear in both the dataset size and dimension. Not surprisingly, IGD has recently become popular in the Web-data and large-scale learning communities.[6,20]

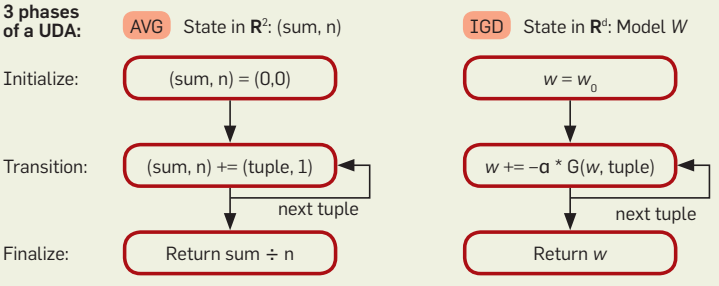
IGD and user-defined aggregates. The key systems insight in Bismarck is that IGD can be implemented using a classic RDBMS abstraction called a UDA (*user-defined aggregate*), which is available in almost every major RDBMS. We prototyped the same Bismarck architecture over PostgreSQL and two commercial RDBMSs. Using a UDA allows Bismarck automatically to leverage mature RDBMS capabilities such as memory management and data marshaling. It also means Bismarck can automatically leverage improvements to the RDBMS as its software evolves.

Figure 8b explains how the core computation in IGD maps to a UDA by comparing it with a SQL AVG. The `state` is the context of aggregation (the model in IGD). The data is a database tuple. The UDA has three stan-

Figure 8. (a) Gradient descent on a convex function; (b) Comparing AVG and IGD as a UDA.



(a)



(b)

Figure 9. Snippets of C-code implementations.

```
LR_Transition(ModelCoef *w, Example e) {
  wx = Dot_Product(w, e.x);
  sig = Sigmoid(-wx * e.y);
  c = stepsize * e.y * sig;
  Scale_And_Add(w, e.x, c); ... }
```

```
SVM_Transition(ModelCoef *w, Example e) {
  wx = Dot_Product(w, e.x);
  c = stepsize * e.y;
  if(1 - wx * e.y > 0) {
    Scale_And_Add(w, e.x, c); } ... }
```

dard functions:

- `Initialize(state)` initializes the model with given values (for example, zeros, or a previous model).
- `Transition(state, data)` automatically executes on each tuple. This is where the core logic (objective function and gradient computations) of the various analytics techniques lies. Thus, the main differences between the implementations of various techniques occur primarily in a few lines of code here (Figure 9), to be explained shortly. Most of the rest of the architecture is common across techniques. This can help reduce development overhead of in-database analytics in Bismarck, in contrast to most existing systems that have a different code architecture for each technique.
- `Finalize(state)` returns the model, possibly persisting it.

A key difference of IGD from aggregates such as AVG is that IGD may need multiple passes (called *epochs*[7]) over the dataset to reach an optimum solution. The number of epochs needed is either given or determined using heuristic convergence tests based on the objective function or gradient's value.[3] Another detail is that Bismarck may randomly reorder the data to improve the convergence rate of IGD.

With Bismarck's unified architecture we could rapidly implement and evaluate four popular analytics techniques—LR (logistic regression), SVM (support vector machine), LMF (low-rank matrix factorization), and CRF (conditional random field)—over three RDBMSes in less than two man-months. This is because, as mentioned earlier, a large fraction of the code infrastructure is common and reusable (on a given RDMBS). For example, starting with a full implementation of LR in Bismarck (in C, over PostgreSQL), fewer than two dozen lines of code need to change to add SVM. (The code, datasets, and a virtual machine with Bismarck preinstalled are available for download: http://hazy.cs.wisc.edu/victor/bismarck-download/.) Figure 9 shows a code snippet comparison of the `Transition` steps of LR and SVM, where the main differences lie. Here, `w` is the coefficient vector, and `e` is a training example with feature vector `x` and label `y`. `Scale _ And _ Add` updates `w` by adding to it `x` multiplied by the scalar `c`. Note the minimal differences between the two implementations.

Similarly, more sophisticated tasks such as LMF were added with only five dozen new lines of code. This is possible since Bismarck abstracts out the invariants of the implementations of the various techniques into a small number of generic functions. This is in contrast to most existing tools, where there is usually a dedicated code stack for each technique. Apart from reducing the development overhead, the simplicity and reusability of Bismarck's architecture enables generic systems-level performance optimizations that apply to many analytics techniques. They enable Bismarck to achieve competitive (often superior) performance against many state-of-the-art commercial and open source tools on many tasks. More importantly, Bismarck achieves automatic scalability to large-scale data, as explained in the next section.

Scalability. For big-data applications, scalability is a central challenge, but Bismarck's architecture is able to achieve scalability seamlessly. Recall that Bismarck makes full use of the powerful RDMBS abstraction of a user-defined aggregate (Figure 8b). The UDA mechanism is an industry standard that has matured over decades of development in RDBMS infrastructure. On a single node, UDAs can scale to as much data as the disk(s) can hold (hundreds of gigabytes on modern machines), but UDAs are also scalable to a parallel database cluster with one addition to the three-function abstraction of Figure 8b: a `Merge(state,state)` function that merges partial aggregates computed on partitioned data in shared-nothing nodes. Although IGD is not algebraic[11] as SQL AVG, we can leverage model-averaging ideas from the literature.[21] Thus, Bismarck offers automatic scalability for many analytics techniques in its unified architecture. In contrast, in many custom-built systems, scalability is either not taken into consideration at all, or the developers will have to worry about implementing data management and scalability issues, often reinventing the wheel.

The accompanying table shows some experimental results validating Bismarck's scalability. (A more comprehensive experimental evaluation is available in the Bismarck paper.[10]) A check mark means the task completes, and X means the approach either crashes or takes longer than 48 hours. N/A means the task is not supported. We compare Bismarck over PostgreSQL against the native analytics tool of a commercial engine DBMS A, as well as

Bismarck scalability.

Task	Dataset			Bismarck PostgreSQL	DBMS A (Native)	In-memory Tools
	Name	#Examples	Size			
LR	Classify300M	300M	135GB	✓	✓	X
SVM				✓	✓	X
LMF	Matrix5B	5B	190GB	✓	N/A	X
CRF	DBLP	2.3M	7.2GB	✓	N/A	X

popular task-specific in-memory tools (Weka, SVMPerf, CRF++, Mallet). All in-memory tools crashed either due to insufficient memory (Weka, SVMPerf, CRF++) or did not terminate even after 48 hours due to thrashing (Mallet). All of the in-RDBMS tools can scale on the simple tasks LR and SVM (less than an hour per epoch for Bismarck), and Bismarck also scales on more complex tasks that are not currently available in DBMS A.

HogWild! and Jellyfish: Extending the infrastructure abstraction. An infrastructure abstraction such as Bismarck's enables us to study generic system optimizations that apply to many techniques. One such optimization is the HogWild! mechanism to parallelize IGD.[15] Modern machines (say, in enterprise settings) typically have multiple cores and shared memory accessible by each core. IGD can then run in parallel on these cores, with the model residing in shared memory.

While one might think locking is required to avoid race conditions, the HogWild! approach is not to lock at all. Under some assumptions, HogWild! guarantees convergence and similar quality. Such lockfree parallelism means HogWild! can achieve near-linear speedups on all the analytics techniques in Figure 7b. We also integrated the HogWild! idea into Bismarck as an alternative to the native UDA parallelism (shared-nothing) and found the former to be significantly faster for large data that can reside on a single node.[10]

Some analytics techniques have specific structures, which can be further exploited to improve performance. Matrix factorization is an excellent example, where the Jellyfish mechanism exploits the structure and outperforms HogWild!. Jellyfish achieves this speedup by using the Latin square pattern to chunk the data matrix.[17] Unlike HogWild!, such chunking enables Jellyfish to run the factorization in parallel on multiple cores without any concurrent overwrites or averaging needed on the model.

Overall, as our experiences with Bismarck, HogWild!, and Jellyfish (and its successor HotTopixx[5]) show, fundamental infrastructure abstractions simplify the development of trained systems by decoupling the algorithms from their implementations. Such decoupling allows us to leverage existing mature code infrastructures such as UDAs, automatically providing features such as maintainability and scalability. It also allows us to devise new performance optimizations that can be designed once and applied to many techniques, rather than reinventing the wheel again and again. While we have focused on applications that use an RDBMS here, the infrastructure abstraction offered by Bismarck is also amenable to newer data platforms such as MapReduce/Hadoop that offer UDA-like aggregation capabilities. We are working on applying our lessons from Bismarck to some of these data platforms as well.

For big-data applications, scalability is a central challenge, but Bismarck's architecture is able to achieve scalability seamlessly.

Future Work and Open Challenges

The Hazy abstractions are a continuously evolving and growing collection, and the primary source of motivation and inspiration is our own experience in developing and deploying trained systems such as GeoDeepDive and DeepDive. As we refine existing abstractions and explore new ones, the following challenges may be particularly interesting (and we are actively working in these directions):

Feature engineering. Conventional wisdom goes that "more signals beat sophisticated models," and our experience in developing GeoDeepDive affirms this idea. Thus, features (or statistical signals) could have a first-class-citizen status just as algorithms in a framework such as Hazy. In contrast to algorithms that are typically off-the-shelf, the effectiveness of features are usually application dependent. As a result, the process of feature engineering tends to be iterative and have humans in the loop. We are trying to abstract feature engineering as a cyclic process involving E3: (data) exploration, (feature) extraction, and (results) evaluation.

Assisted development. Traditionally, developing trained systems requires expertise, experience, and a deep understanding of the data and algorithms. As a result, usually only a small number of developers would feel qualified for such applications; and the development process would often be tricky or painstaking even for these developers. To lower the barrier and

improve the productivity of developing such applications, we are exploring various options to support assisted development (for example, automatic feature suggestion, automatic parameter tuning, and smart diagnosis of trained systems[2]).

New data platforms. Some recent projects aim to bring statistical tools to the Hadoop environment.[4,9] It would be interesting to port the Hazy abstractions to Hadoop (and associated systems such as HBase and Accumulo). The resulting combination is likely to enjoy a large and active user base of developers. A key challenge in this direction is how to reconcile the roles played by the RDBMS in Hazy in the Hadoop environment. We are exploring possible solutions to address this challenge.

Conclusion

There is a race to unleash the full potential of big data using statistical and machine-learning techniques. The high-profile success of many recent big-data analytics-driven systems, also known as trained systems, has generated great interest in bringing such technological capabilities to a wider variety of domains. A key challenge in converting this potential to reality is making these trained systems easier to build and maintain.

The Hazy project outlined an approach to tackling this challenge by identifying common patterns, also known as abstractions, in building such systems. The abstractions allow for decoupling the concerns of applications from the algorithms that are used and the underlying implementations that are needed. Optimizing and supporting these abstractions as primitives enables us to make it easier to build and maintain trained systems. Our ideas are shaped by, and continue to evolve with, our own experiences with building such systems (DeepDive, GeoDeepDive, and AncientText), as well as our repeated interactions with practitioners at major enterprises and developers at major analytics companies.

The Hazy Research Group's philosophy is to make all our research software available as open source. The source code, installation and usage documentation, datasets used for our research publications, and virtual machines with the software tools and datasets preinstalled are available at http://hazy.cs.wisc.edu. The tools have already been downloaded thousands of times. Videos providing overviews of Hazy projects and tutorials are available at http://www.youtube.com/user/HazyResearch/videos. Hazy code has been adopted and shipped into production by four companies, is used by many other research groups, and has been deployed at a scientific observatory at the South Pole.

Acknowledgments

The Hazy Research Group is a team of Ph.D., M.S., and undergraduate students, working under the supervision of Christopher Ré. The Hazy project is made possible due to the endless enthusiasm and tireless efforts of these students. In addition to the authors, the following students also contributed to this article: Victor Bittorf, Xixuan Feng, and Ce Zhang. We gratefully acknowledge the support of DARPA grant FA8750-09-C-0181, NSF CAREER award IIS1054009, ONR award N000141210041, and gifts or research awards from American Family Insurance, Google, Greenplum, Johnson Controls, Inc., LogicBlox, Oracle, and Raytheon. We appreciate the support of the Center for High Throughput Computing and Miron Livny's Condor research group. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the view of the above companies, DARPA, or the U.S. government.

Related articles on queue.acm.org

The Pathologies of Big Data
Adam Jacobs
http://queue.acm.org/detail.cfm?id=1563874

How Will Astronomy Archives Survive the Data Tsunami?
G. Bruce Berriman, Steven L. Groom
http://queue.acm.org/detail.cfm?id=2047483

Condos and Clouds
Pat Helland
http://queue.acm.org/detail.cfm?id=2398392

References

1. Agrawal, R. and Srikant, R. Fast algorithms for mining association rules in large databases. In *Proceedings of Very Large Databases*, 1994.
2. Anderson, M., Antenucci, D., Bittorf, V., Burgess, M., Cafarella, M., Kumar, A., Niu, F., Park, Y., Ré, C. and Zhang, C. 2013. Brainwash: A data system for feature engineering. In *Proceedings of Conference on Innovative Data Systems Research*, 2013
3. Anstreicher, K.M., Wolsey, L.A. Two "well-known" properties of subgradient optimization. *Mathematical Programming 120*, 1 (2009), 213–220.
4. Apache Mahout; http://mahout.apache.org/.
5. Bittorf, V., Recht, B., Ré, C. and Tropp, J. Factoring nonnegative matrices with linear programs. In *Proceedings of Neural Information Processing Systems*, 2012.
6. Bottou, L. and Bousquet, O. The tradeoffs of large scale learning. In *Proceedings of Neural Information Processing Systems*, 2007.
7. Bottou, L. and LeCun, Y. Large scale online learning. In *Proceedings of Neural Information Processing Systems*, 2003.
8. Boyd, S. and Vandenberghe, L. *Convex Optimization*. Cambridge University Press, NY, 2004.
9. Das, S., Sismanis, Y., Beyer, K. S., Gemulla, R., Haas, P. J. and McPherson, J. Ricardo: Integrating R and Hadoop. In *Proceedings of ACM SIGMOD*, 2010.
10. Feng, X., Kumar, A., Recht, B. and Ré, C. Towards a unified architecture for in-RDBMS analytics. In *Proceedings of ACM SIGMOD*, 2012.
11. Gray, J., Chaudhuri, S., Bosworth, A., Layman, A., Reichart, D., Venkatrao, M., Pellow, F., and Pirahesh, H. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Mining and Knowledge Discovery 1*, 1 (1997).
12. Hastie, T., Tibshirani, R. and Friedman, J.H. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer-Verlag, NY, 2011.
13. Hellerstein, J., Ré, C., Schoppmann, F., Wang, D. Z., Fratkin, E., Gorajek, A., Ng, K. S., Welton, C., Feng, X., Li, K. and Kumar, A. The MADlib Analytics Library or MAD Skills, the SQL. In *Proceedings of the VLDB Endowment 5*, 12 (2012): 1700–1711.
14. Niu, F., Ré, C., Doan, A. and Shavlik, J. Tuffy: Scaling up statistical inference in Markov logic networks using an RDBMS. In *Proceedings of Very Large Databases*, 2011.
15. Niu, F., Recht, B., Ré, C. and Wright, S. Hogwild!: a lock-free approach to parallelizing stochastic gradient descent. In *Proceedings of Neural Information Processing Systems*, 2011.
16. Niu, F., Zhang, C., Ré, C. and Shavlik, J. Elementary: Large-scale knowledge-base construction via machine learning and statistical inference. *International Journal on Semantic Web and Information Systems-Workshop on Web-scale Knowledge Extraction*, 2012.
17. Recht, B. and Ré, C. Parallel stochastic gradient algorithms for large-scale matrix completion. In *Optimization Online*, 2012.
18. Richardson, M. and Domingos, P. Markov logic networks. *Machine Learning 62* (2006), 107–136.
19. Rockafellar, R.T. *Convex Analysis* (Princeton Landmarks in Mathematics and Physics). Princeton University Press, Princeton, NJ, 1996.
20. Vowpal Wabbit; http://hunch.net/~vw/.
21. Zinkevich, M., Weimer, M., Smola, A. and Li, L. Parallelized stochastic gradient descent. In *Proceedings of Neural Information Processing Systems*, 2010.

Arun Kumar is a Ph.D. student at the University of Wisconsin-Madison. His research interests are in the areas of data management, with a focus on data analytics and managing uncertain data. He received his M.S. from the University of Wisconsin-Madison in 2011, and his BTech from the Indian Institute of Technology-Madras in 2009, both in computer science.

Feng Niu is a software engineer at Google, Inc. His goal is to help the machine help organize the world's information with more structures, connections, and insights, but with less human effort. In 2012, he received his Ph.D. in computer science from the University of Wisconsin-Madison. His graduate study was mainly funded by the DARPA Machine Reading program. He received his BE degree in Computer Science from Tsinghua University in 2008.

Christopher (Chris) Ré is an assistant professor in the department of computer sciences at the University of Wisconsin-Madison. The goal of his work is to enable users and developers to build applications that more deeply understand and exploit data. He received his Ph.D. from the University of Washington, Seattle; his work in the area of probabilistic data management received the SIGMOD 2010 Jim Gray Dissertation Award. Ré received an NSF CAREER Award in 2011.