Towards a Unified Architecture for in-RDBMS Analytics

Xixuan Feng Arun Kumar Benjamin Recht Christopher Ré

Department of Computer Sciences
University of Wisconsin-Madison
{xfeng, arun, brecht, chrisre}@cs.wisc.edu

ABSTRACT

The increasing use of statistical data analysis in enterprise applications has created an arms race among database vendors to offer ever more sophisticated in-database analytics. One challenge in this race is that each new statistical technique must be implemented from scratch in the RDBMS, which leads to a lengthy and complex development process. We argue that the root cause for this overhead is the lack of a unified architecture for in-database analytics. Our main contribution in this work is to take a step towards such a unified architecture. A key benefit of our unified architecture is that performance optimizations for analytics techniques can be studied generically instead of an ad hoc, per-technique fashion. In particular, our technical contributions are theoretical and empirical studies of two key factors that we found impact performance: the order data is stored, and parallelization of computations on a single-node multicore RDBMS. We demonstrate the feasibility of our architecture by integrating several popular analytics techniques into two commercial and one open-source RDBMS. Our architecture requires changes to only a few dozen lines of code to integrate a new statistical technique. We then compare our approach with the native analytics tools offered by the commercial RDBMSes on various analytics tasks, and validate that our approach achieves competitive or higher performance, while still achieving the same quality.

Categories and Subject Descriptors

H.2 [**Database Management**]: Miscellaneous

Keywords

Analytics, Convex Programming, Incremental Gradient Descent, User-Defined Aggregate

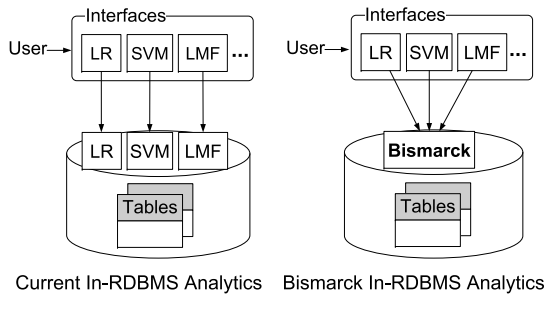
1. INTRODUCTION

There is an escalating arms race to bring sophisticated data analysis techniques into enterprise applications. In the late 1990s and early 2000s, this brought a wave of data mining toolkits into the RDBMS. Several major vendors are again making an effort toward sophisticated in-database analytics with both open source efforts, e.g., the MADlib platform [17], and several projects at major database vendors. In our conversations with engineers from Oracle [38] and EMC Greenplum [21], we learned that a key bottleneck in this arms race is that each new data analytics technique requires several ad hoc steps: a new solver is employed that has new memory requirements, new data access methods, etc. As a result, there is little code reuse across different algorithms, slowing the development effort. Thus, it would be a boon to the database industry if one could devise a *single architecture* that was capable of processing many data analytics techniques. An ideal architecture would leverage as many of the existing code paths in the database as possible as such code paths are likely to be maintained and optimized as the RDBMS code evolves to new platforms.

To find this common architecture, we begin with an observation from the mathematical programming community that has been exploited in recent years by both the statistics and machine learning communities: many common data analytics tasks can be framed as *convex programming problems* [16, 26]. Examples of such convex programming problems include support vector machines, least squares and logistic regression, conditional random fields, graphical models, control theoretic models, and many more. It is hard to overstate the impact of this observation on data analysis theory: rather than studying properties of each new model, researchers in this area are able to unify their algorithmic and theoretical studies. In particular, convex programming problems are attractive as local solutions are always globally optimal, and one can find local solutions via a standard suite of well-established and analyzed algorithms. Thus, convex programming is a natural starting point for a unified analytics architecture.

The mathematical programming literature is filled with algorithms to solve convex programming problems. Our first goal is to find an algorithm in that literature whose data access properties are amenable to implementation inside an RDBMS. We observe that a classical algorithm from the mathematical programming cannon, called *incremental gradient descent* (IGD), has a data-access pattern that is essentially identical to the data access pattern of any SQL aggregation function, e.g., an SQL AVG. As we explain in Section 2, IGD can be viewed as examining the data one tuple at time and then performing a (non-commutative) aggregation of the results. Our first contribution is an archi-

SIGMOD '12, May 20–24, 2012, Scottsdale, Arizona, USA.



Analytics Task	Objective
Logistic Regression (LR)	$\sum_i \log(1 + exp(-y_i w^T x_i)) + \mu\|\vec{w}\|_1$
Classification (SVM)	$\sum_i (1 - y_i w^T x_i)_+ + \mu\|\vec{w}\|_1$
Recommendation (LMF)	$\sum_{(i,j)\in\Omega} (L_i^T R_j - M_{ij})^2 + \mu\|L, R\|_F^2$
Labeling (CRF) [48]	$\sum_k \left[\sum_j w_j F_j(y_k, x_k) - \log Z(x_k)\right]$
Kalman Filters	$\sum_{t=1}^{T} \|\|Cw_t - f(y_t)\|\|_2^2 + \|\|w_t - Aw_{t-1}\|\|_2^2$
Portfolio Optimization	$p^T w + w^T \Sigma w$ s.t. $w \in \Delta$

Figure 1: BISMARCK in an RDBMS: (A) In contrast to existing in-RDBMS analytics tools that have separate code paths for different analytics tasks, BISMARCK provides a single framework to implement them, while possibly retaining similar interfaces. (B) Example tasks handled by BISMARCK. In Logistic Regression and Classification, we minimize the error of a predictor plus a regularization term. In Recommendation, we find a low-rank approximation to a matrix M which is only observed on a sparse sampling of its entries. This problem is not convex, but it can still be solved via IGD. In Labeling with Conditional Random Fields, we maximize the weights associated with features (F_j) in the text to predict the labels. In Kalman Filters, we fit noisy time series data. In quantitative finance, portfolios are optimized balancing risk ($p^T w$) with expected returns ($w^T \Sigma w$); the allocations must lie in a simplex, Δ, i.e., $\Delta = \{w \in \mathbb{R}^n \mid \sum_{i=1}^n w_i = 1\}$ and $w_i \geq 0$ for $i = 1, \ldots, n$.

tecture that leverages this observation: *we show that we can implement these methods using the user-defined aggregate features that are available inside every major RDBMS*. To support our point, we implement our architecture over PostgreSQL and two commercial database systems. In turn, this allows us to implement all convex data analysis techniques that are available in current RDBMSes – and many next generation techniques (see Figure 1). The code to add in a new model can be as little as ten lines of C code, e.g., for logistic regression.[1]

As with any generic architectural abstraction, a key question is to understand how much performance overhead our approach would incur. In the two commercial systems that we investigate, we show that compared to a strawman user-defined aggregate that computes no value, our approach has between 5% (for simple tasks like regression) to 100% overhead (for complex tasks like matrix factorization). What is perhaps more surprising is that our approach is often much faster than existing in-database analytic tools from commercial vendors: our prototype implementations are in many cases $2-4$x faster than existing approaches for simple tasks – and for some newly added tasks such as matrix factorization, orders of magnitude faster.

A second benefit of a unified in-database architecture is that we can study the factors that impact performance and optimize them in a way that applies across several analytics tasks. Our preliminary investigation revealed many such optimization opportunities including data layout, compression, data ordering, and parallelism. Here, we focus on two such factors that we discovered were important in our initial prototype: *data clustering*, i.e., how the data is ordered on-disk, and *parallelism* on a single-node multicore system.

Although IGD will converge to an optimal solution on convex programming problems no matter how the underlying data is ordered, empirically some orders allow us to terminate more quickly than others. We observe that inside an RDBMS, data is often clustered for reasons unrelated to the analysis task (e.g., to support efficient query performance), and running IGD through the data in the order that is stored on disk can lead to considerable degradation in performance. With this in mind, we describe a theoretical example that characterizes some "bad" orders for IGDs and shows that they are indeed likely inside an RDBMS. For example, if one clusters the data for a classification task such that all of the positive examples come before the negative examples, the resulting convergence rate may be much slower than if the data were randomly ordered, i.e., to reach the same distance to the optimal solution, more passes over the data are needed if the data is examined by IGD in the clustered order versus a random order. Our second technical contribution is to describe the clustering phenomenon theoretically, and use this insight to develop a simple approach to combat this. A common approach in machine learning randomly permutes the data with each pass. However, such random shuffling may incur substantial computational overhead. Our method obviates this overhead by shuffling the data only once before the first pass. We implement and benchmark this approach on all three RDBMSes that we study: empirically, we find that across a broad range of models, while shuffling once has a slightly slower convergence rate than shuffling on each pass, the lack of expensive reshuffling allows us to simply run more epochs in the same amount of time. Thus, shuffling once has better overall performance than shuffling always.

We then study how to parallelize IGD in an RDBMS. We first observe that recent work in the machine learning community allows us to parallelize IGD [52] in a way that leverages the standard user-defined aggregation features available in every RDBMS to do shared-nothing parallelism. We leverage this parallelization feature in a commercial database and show that we can get almost linear speed-ups. However, recent results in the machine learning community have shown that these approaches may yield sub-optimal runtime performance compared to approaches that exploit shared-memory parallelism [29, 37]. This motivates us to adapt approaches that exploit shared memory for use inside an RDBMS. We focus on single-node multicore parallelism where shared memory is available. Although not in the textbook description of an RDBMS, all three RDBMSes we inspected allow us to allocate and manage some shared memory (even providing interfaces to help manage the nec-

[1] Not all data analysis problems are convex. Notable exceptions are Apriori [9] and graph mining algorithms.

essary data structures). We show that the shared-memory version converges faster than the shared-nothing version.

In some cases, a single shuffle of the data may be too expensive (e.g., for data sets that do not fit in available memory). To cope with such large data sets, users often perform a subsampling of the data (e.g., using a *reservoir sample* [46]). Subsampling is not always desirable, as it introduces an additional error (increasing the variance of the estimate). Thus, for such large data sets, we would like to avoid the costly shuffle of the data to achieve better performance than subsampling. Our final technical contribution combines the parallelization scheme and *reservoir sampling* to get our highest performance results for datasets that do not fit in available RAM. On simple tasks like logistic regression, we are 4X faster than state-of-the-art in-RDBMS tools. On more complex tasks like matrix factorization, these approaches allow us to converge in a few hours, while existing tools do not finish even after several days.

In summary, our work makes the following contributions:

- We describe a novel unified architecture, BISMARCK, for integrating many data analytics tasks formulated as Incremental Gradient Descent into an RDBMS using features available in almost every commercial and open-source system. We give evidence that our architecture is widely applicable by implementing BISMARCK in three RDBMS engines: PostgreSQL and two commercial engines.
- We study the effect of data clustering on performance. We identify a theoretical example that shows that bad orderings not typically considered in machine learning do occur in databases and we develop a novel strategy to improve performance.
- We study how to adapt existing approaches to make BISMARCK run in parallel. We verify that this allows us to achieve large speed-ups on a wide range of tasks using features in existing RDBMSes. We combine our solution for clustering with the above parallelization schemes to attack the problem of bad data ordering.

We validate our work by implementing BISMARCK on three RDBMS engines: PostgreSQL, and two commercial engines, DBMS A and DBMS B. We perform an extensive experimental validation. We see that we are competitive, and often better than state-of-the-art in-database tools for standard tasks like regression and classification. We also show that for next generation tasks like conditional random fields, we have competitive performance against state-of-the-art special-purpose tools.

Related Work. Every major database vendor has data mining tools associated with their RDBMS offering. Recently, there has been an escalating arms race to add sophisticated analytics into the RDBMS with each iteration bringing more sophisticated tools into the RDBMS. So far, this arms race has centered around bringing individual statistical data mining techniques into an RDBMS, notably Support Vector Machines [35], Monte Carlo sampling [27, 51], Conditional Random Fields [25, 49], and Graphical Models [43, 50]. Our effort is inspired by these approaches, but the goal of this work is to understand the extent to which we can handle these analytics tasks with a single unified architecture. Of these approaches, MCDB [27] and Wick et al. [51] are the most related in that they propose a single unified interface for uncertain data based on sampling and graphical models respectively. In contrast, we consider data analytics techniques that are modeled as convex programming problems.

A related (but orthogonal issue) is how statistical models should be integrated into the RDBMS to facilitate ease of use, notably *model-based views* pioneered in MauveDB [19]. The idea is to give users a unified abstraction that hides from the user (but not the tool developer) the details of statistical processing. In contrast, our goal is a lower level abstraction: we want to unify at the implementation of many different data analysis tasks.

Using incremental gradient algorithms for convex programming problems is a classical idea, going back to the seminal work in the 1950s of Robbins and Monro [40]. Recent years have seen a resurgence of interest in these algorithms due to their ability to tolerate noise, converge rapidly, and achieve high runtime performance. In fact, sometimes an IGD method can converge before examining all of the data; in contrast, a traditional gradient method would need to touch all of the data items to take even a single step. These properties have made IGD an algorithm of choice in the Web community. Notable implementations include Vowpal Wabbit at Yahoo! [7], and in large-scale learning [14]. IGD has also been employed for specific algorithms, notably Gemulla et al recently used it for matrix factorization [23]. What distinguishes our work is that we have observed that IGD forms the basis of a systems abstraction that is well suited for in-RDBMS processing. As a result, our technical focus is on optimizations that are implementable in an RDBMS and span many different models.

Our techniques to study the impact of sorting is inspired by the work of Bottou and LeCun [15], who empirically studied the related problem of different sampling strategies for stochastic gradient algorithms. There has been a good deal of work in the machine learning community to create several clever parallelization schemes for IGD [12, 18, 20, 29, 53]. Our work builds on this work to study those methods that are ideally suited for an RDBMS. For convex programming problems, we find that the model averaging techniques of Zinkevich et al [53] fit well with user-defined aggregates. Recently, work on using shared memory *without locking* has been shown to converge more rapidly in some settings [37]. We borrow from both approaches.

Finally, the area of convex programming problems is a hot topic in data analysis [12, 16], e.g., the support vector machine [32], Lasso [44], and logistic regression [47] were all designed and analyzed in a convex programming framework. Convex analysis also plays a pivotal role in approximation algorithms, e.g., the celebrated MAX-CUT relaxation [24] shows that the optimal approximation to this classical NP-hard problem is achieved by solving a convex program. In fact a recent result in the Theory community shows that there is reason to believe that almost all combinatorial optimization problems have optimal approximations given by solving convex programs [39]. Thus, we argue that these techniques may enable a number of sophisticated data processing algorithms in the RDBMS.

Outline. The rest of the paper is organized as follows: In Section 2, we explain how BISMARCK interacts with the RDBMS, and give the necessary mathematical programming background on gradient methods. In Section 3, we discuss the architecture of BISMARCK, and how data ordering and

parallelism impact performance. In Section 4, we validate that BISMARCK is able to integrate analytics techniques into an RDBMS with low overhead and high performance.

2. PRELIMINARIES

We start with a description of how BISMARCK fits into an RDBMS, and then give a simple example of how an end-user interacts with BISMARCK in an RDBMS. We then discuss the necessary mathematical programming background on gradient methods.

2.1 BISMARCK in an RDBMS

We start by contrasting the high level architecture of most existing in-RDBMS analytics tools with how BISMARCK integrates analytics into an RDBMS, and explain how BISMARCK is largely orthogonal to the *end-user interfaces*. Existing tools like MADlib [17], Oracle Data Mining [4], and Microsoft SQL Server Data Mining [1] provide SQL-like interfaces for the end-user to specify tasks like Logistic Regression, Support Vector Machine, etc. Declarative interface-level abstractions like *model-based views* [19] help in creating such user-friendly interfaces. However, the underlying implementations of these tasks do not have a unified architecture, increasing the overhead for the developer. In contrast, BISMARCK provides a single architectural abstraction for the developer to unify the *in-RDBMS implementations* of these analytics techniques, as illustrated in Figure 1. Thus, BISMARCK is orthogonal to the end-user interface, and the developer has the freedom to provide any existing or new interfaces. In fact, in our implementation of BISMARCK in each RDBMS, BISMARCK's user-interface mimics the interface of that RDBMS' native analytics tool.

For example, consider the interface provided by the open-source MADlib [17] used over PostgreSQL and Greenplum databases. Consider the task of classifying papers using a support vector machine (SVM). The data is in a table `LabeledPapers(id, vec, label)`, where `id` is the key, `vec` is the feature values (say as an array of floats) and `label` is the class label. In MADlib, the user can train an SVM model by simply issuing a SQL query with some pre-defined functions that take in the data table details, parameters for the model, etc. [17] In BISMARCK, we mimic this familiar interface for users to do in-RDBMS analytics. For example, the query (similar to MADlib's) to train an SVM is as follows:

```
SELECT SVMTrain ('myModel', 'LabeledPapers',
                 'vec', 'label');
```

SVMTrain is a function that passes the user inputs to BISMARCK, which then performs the gradient computations for SVM and returns the model. The model, which is basically a vector of coefficients for an SVM, is then persisted as a user table `'myModel'`. The model can be applied to new unlabeled data to make predictions by using a similar SQL query.

2.2 Background: Gradient Methods

We provide a brief introduction to gradient methods. For a thorough introduction to gradient methods and their projected, incremental variants, we direct the interested reader to the many surveys of the subject [13, 36]. We focus on a particular class of problems that have *linearly separable* objective functions. Formally, our goal is to find a vector $w \in \mathbb{R}^d$ for some $d \geq 1$ that minimizes the following objective:[2]

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

Here, the objective function decomposes into a sum of functions $f(w, z_i)$ for $i = 1, \ldots, N$ where each z_i is an item of (training) data. In BISMARCK, the z_i are represented by tuples in the database, e.g., a pair (paper,area) for paper classification. We abbreviate $f(w, z_i) = f_i(w)$. For example, in SVM classification, the function $f_i(w)$ could be the hinge loss of the model w on the ith data element and $P(w)$ enforces the smoothness of the classifier (preventing overfitting). Eq. 1 is general: Figure 1(B) gives an incomplete list of examples that can be handled by BISMARCK.

A gradient is a generalization of a derivative that tells us if the function is increasing or decreasing as we move in a particular direction. Formally, a *gradient* of a function $h : \mathbb{R}^d \to \mathbb{R}$ is a function $\nabla h : \mathbb{R}^d \to \mathbb{R}^d$ such that $(\nabla h(w))_i = \frac{\partial}{\partial w_i} h(w)$ [16]. Linearity of the gradient implies the equation:

$$\nabla \sum_{i=1}^{N} f_i(w) = \sum_{i=1}^{N} \nabla f_i(w)\,.$$

For our purpose, the importance of this equation is that to compute the gradient of the objective function, we can compute the gradient of each f_i individually.

Gradient methods are algorithms that solve (1). These methods are defined by an iterative rule that describes how one produces the $(k+1)$-st iterate, $w^{(k+1)}$, given the previous iterate, $w^{(k)}$. For simplicity, we assume that $P = 0$. Then, we are minimizing a function $f(w) = \sum_{i=1}^{N} f_i(w)$, our goal is to produce a new point $w^{(k+1)}$ where $f(w^{(k)}) > f(w^{(k+1)})$. In 1-D, we need to move in the direction opposite the derivative (gradient). A gradient method is defined by the rule:

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

here $\alpha_k \geq 0$ is a positive parameter called *step-size* that determines how far to follow the current search direction. Typically, $\alpha_k \to 0$ as $k \to \infty$.

The twist for *incremental* gradient methods is to approximate the full gradient using a single terms of the sum. That is, let $\eta(k) \in \{1, \ldots, N\}$, chosen at iteration k. Intuitively, we approximate the gradient $\nabla f(w)$ with $\nabla f_{\eta(k)}(w)$.[3] Then,

$$w^{(k+1)} = w^{(k)} - \alpha_k \nabla f_{\eta(k)}(w^{(k)}) \tag{2}$$

This is a key connection: each f_i can be represented as a single tuple. We illustrate this rule with a simple example:

EXAMPLE 2.1. *Consider a simple least-squares problem with $2n$ ($n \geq 1$) data points $(x_1, y_1), \ldots, (x_{2n}, y_{2n})$. The feature values are $x_i = 1$ for $i = 1, \ldots, 2n$ and the labels are $y_i = 1$ for $i \leq n$, and $y_i = -1$, otherwise. The resulting mathematical programming problem is:*

$$\min_{w} \quad \frac{1}{2} \sum_{i=1}^{2n} (w x_i - y_i)^2$$

[2] In Appendix A, we generalize to include constraints via proximal point methods. One can also generalize to both matrix valued w and non-differentiable functions [42].

[3] Observe that minimizing f and $g(w) = \frac{1}{N} f(w)$, means correcting by the factor N is not necessary and not done by convention.

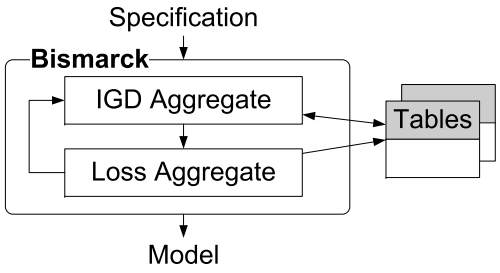


Figure 2: High-level Architecture of BISMARCK.

Since $x_i = 1$ for all i, the optimal solution to the problem is the mean $w = 0$, but we choose this to illustrate the mechanics of the method. We begin with some point $w^{(0)}$ chosen arbitrarily. We choose $i \in \{1, \ldots, 2n\}$ at random. Fix some $\alpha \geq 0$ and for $k \geq 0$, set $\alpha_k = \alpha$ for simplicity. Then, our approximation to the gradient is $\nabla f_i(w^{(0)}) = (w^{(0)} - y_i)$. And so, our first step is:

$$w^{(1)} = w^{(0)} - \alpha(w^{(0)} - y_i)$$

We then repeat the process with $w^{(2)}$, etc. One can check that after $k+1$ steps, we will have:

$$w^{(k+1)} = (1-\alpha)^{k+1} w_0 + \alpha \sum_{j=0}^{k} (1-\alpha)^{k-i} y_{\eta(j)}$$

Since the expectation of $y_{\eta(j)}$ equals 0, we can see that we converge exponentially quickly to 0 under this scheme – even before we see all $2n$ points. This serves to illustrate why an IGD scheme may converge much faster than traditional gradient methods, where one must touch every data item at least once just to compute the first step.

Remarkably, when both the functions $\sum_{i=1}^{n} f_i(w)$ and $P(w)$ are both convex, the incremental gradient method is guaranteed to converge to a globally optimal solution [36] at known rates. Also, IGD converges (perhaps at a slower rate) even if $\eta(k)$ is a sequence in a fixed, arbitrary order [11,30,31,33,45]. We explore this issue in more detail in Example 3.1.

3. BISMARCK ARCHITECTURE

We first describe the high-level architecture of BISMARCK, and then explain how we implement IGD in an RDBMS. Then, we drill down into two aspects of our architecture that impact performance - *data ordering* and *parallelism*.

3.1 High-Level Architecture

The high-level architecture of BISMARCK is presented in Figure 2. BISMARCK takes in the specifications for an analytics task (e.g., data details, parameters, etc.) and runs the task using Incremental Gradient Descent (IGD). As explained before, IGD allows us to solve a number of analytics tasks in one unified way. The main component of BISMARCK is the in-RDBMS implementation of IGD with a data access pattern similar to a SQL aggregate query. For this purpose, we leverage the mechanism of User-Defined Aggregate (UDA), a standard feature available in almost all RDBMSes [2, 3, 5]. The UDA mechanism is used to run the IGD computation, but also to test for convergence and compute information, e.g., error rates. BISMARCK also needs to provide a simple iteration to test for convergence. We will explain more about these two aspects shortly, but first we describe the architecture of a UDA, and how we can handle IGD in this framework.

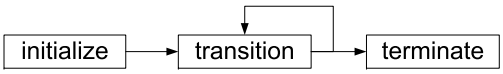


Figure 3: The Standard Three Phases of a UDA.

IGD as a User-Defined Aggregate. As shown in Figure 3, a developer implements a UDA by writing three standard functions: `initialize(state)`, `transition(state, data)` and `terminate(state)`. Almost all RDBMSes provide the abstraction of a UDA, albeit with different names or interfaces for these three steps, e.g., PostgreSQL names them 'initcond', 'sfunc' and 'finalfunc' [5].

The `state` is basically the context of aggregation (e.g., the running total and count for an `AVG` query). The `data` is a tuple in the table. In our case, the `state` is essentially the *model* (e.g., the coefficients of a logistic regressor) and perhaps some meta data (e.g., number of gradient steps taken). In our current implementation, we assume that the `state` fits in memory (models are typically orders of magnitude smaller than the data, which is not required to fit in memory). The `data` is again an example from the data table, which includes the attribute values and the label (for supervised schemes). We now explain the role of each function:

- The `initialize(state)` function initializes the model with user-given values (e.g., a vector of zeros), or a model returned by a previous run.
- In `transition(state, data)`, we first compute the (incremental) gradient value of the objective function on the given `data` example, and then update the current model (Equation 2 from Section 2.2). This function is where one puts the logic of the the various analytics techniques – each technique has its own objective function and gradient (Figure 1(B)). Thus, the main differences in the implementations of the various analytics techniques occur mainly in a few lines of code within this function, while the rest of our architecture is reused across techniques. Figure 4 illustrates the claim with actual code snippets for two tasks (LR and SVM). This simplifies the development of sophisticated in-database analytics, in contrast to existing systems that usually have different code paths for different techniques (Figure 1(A)).
- In `terminate(state)`, we finish the gradient computations and return the model, possibly persisting it.

A key implementation detail is that BISMARCK may reorder the data to improve the convergence rate of IGD or to sample from the data. This feature is supported in all major RDBMSes, e.g., in PostgreSQL using the `ORDER BY RANDOM()` construct.

Key Differences: Epochs and Convergence. A key difference from traditional aggregations, like SUM, AVG, or MAX, is that to reach the optimal objective function value, IGD may need to do more than one pass over the dataset.

```
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); } ... }
```

Figure 4: Snippets of the C-code implementations of the `transition` step for Logistic Regression (LR) and Support Vector Machine (SVM). 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.

Following the machine learning literature, we call each pass an *epoch* [15]. Thus, the aggregate may need to be executed more than once, with the output model of one run being input to the next (shown in Figure 2 as a loop). To determine how many epochs to run, BISMARCK supports an arbitrary Boolean function to be called (which may itself involve aggregation). This supports both what we observed in practice as common heuristic convergence tests, e.g., run for a fixed number of iterations, and more rigorous conditions based on the norm of the gradient common in machine learning [10].

A second difference is that the we may need to compute the actual value of the objective function (also known as the *loss*) using the model after each epoch. The loss value may be needed by the stopping condition, e.g., a common convergence test is based on the relative drop in the loss value. This loss computation can also be implemented as a UDA (or piggybacked onto the IGD UDA).

Technical Opportunities. A key conceptual benefit of BISMARCK's approach is that one can study *generic* performance optimizations (i.e., optimizations that apply to many analytics techniques) rather than ad hoc, per-technique ones. The remainder of the technical sections are devoted to examining two such generic optimizations. First, the conventional wisdom is that for IGD to converge more rapidly, each data point should be sampled in random (without-replacement) order [15]. This can be achieved by randomly reordering, or *shuffling*, the dataset before running the aggregate for gradient computation at each epoch. The goal of course is to converge faster in wall-clock time, not per epoch. Thus, we study when the increased speed in convergence rate per epoch outweighs the additional cost of reordering the data at each epoch. The second optimization we describe is how to leverage multicore parallelism to speed-up the IGD aggregate computation.

3.2 Impact of Data Ordering

On convex programming problems, IGD is known to converge to the optimal value irrespective of how the underlying data is ordered. But empirically some data orderings allow us to converge in fewer epochs than others. However, our experiments suggest that the sensitivity is not as great as one might think. In other words, presenting the data in a random order gets essentially optimal run-time behavior. This begs the question as to whether we should even reorder the data randomly at each epoch. In fact, some machine learning tools do not even bother to randomly reorder the data. However, we observe that inside an RDBMS, data is often clustered for reasons unrelated to the analysis task (e.g., for efficient join query performance). For example, the data for a classification task might be clustered by the class label. We now analyze this issue by providing a theoretical example that characterizes pathological orders for IGD. We chose this example to illustrate the important points with respect to clustering and be as theoretically simple as possible.

EXAMPLE 3.1 (1-D CA-TX). *Suppose that our data is clustered geographically, e.g., sales data from California, followed by Texas, and the attributes of the sales in the two states cause the data to be in two different classes. With this in mind, recall Example 2.1. We are given a simple least-squares problem with $2n$ ($n \geq 1$) data points (x_1, y_1), ..., (x_{2n}, y_{2n}). The feature values are $x_i = 1$ for $i = 1, \ldots, 2n$ and the labels are $y_i = 1$ for $i \leq n$, and $y_i = -1$, otherwise. The resulting mathematical programming problem is:*

$$\min_w \quad \frac{1}{2}\sum_{i=1}^{2n}(wx_i - y_i)^2$$

Since $x_i = 1$ for all i, the optimal solution is the mean, $w = 0$. But our goal here is to analyze the behavior of IGD on this problem under various orders. Due to this problem's simplicity, we can solve the behavior of the resulting dynamical system in closed form under a variety of ordering schemes. Consider two schemes to illustrate our point: (1) data points seen are randomly sampled from the dataset, and (2) data points seen in ascending index order, (x_1, y_1), (x_2, y_2), ... Scheme (2) simulates operating on data that is clustered by class.

Figure 5 plots the value of w during the course of the IGD under the above two sampling schemes (using diminishing step-size rule). We see that both approaches do indeed converge to the optimal value, but approach (1), which uses random sampling, converges more rapidly. In contrast, in approach (2), w oscillates between 1 and -1, until converging eventually. Intuitively, this is so because the IGD initially takes steps influenced by the positive examples, and is later influenced by the negative examples (within one epoch). In other words, convergence can be much slower on clustered data. In the extended version of the paper [22], we present calculations to precisely explain this behavior. We conclude the example by noting that almost all *permutations of the data will behave similar to (1), and not (2). In other words, (2) is a pathological ordering, but one which is indeed possible for data stored in an RDBMS.*

Shuffling the data at each epoch is expensive and incurs a high overhead. In fact, for simple tasks like LR and SVM, the shuffling time dominates the gradient computation time by a factor of 5. To remove the overhead of shuffling the data at every epoch, while still avoiding the pathological ordering, we propose a simple solution – shuffle the data *only once*. By randomly reordering the data once, we avoid the pathological ordering that might be present in data stored in a database. We implemented and benchmarked this approach on all three RDBMSes that we study. As explained

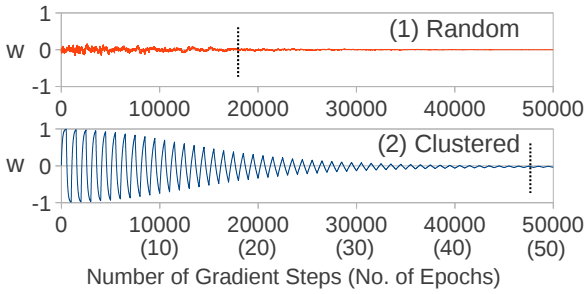


Figure 5: 1-D CA-TX Example: Plot of w against number of gradient steps on (1) *Random*, and (2) *Clustered* data orderings for a dataset with 1000 examples (i.e., $n = 500$). The number of epochs is shown in parentheses on the x-axis. *Random* takes 18 epochs to converge (convergence defined here as $w^2 < 0.001$), while *Clustered* takes 48 epochs.

later in Section 4.3, empirically, we find that shuffling once suffices across a broad range of models. Shuffling once does have a slightly lower convergence rate than shuffling always. However, since we need not shuffle at every epoch, we significantly reduce the runtime per epoch, which means we can simply run more epochs within the same wall-clock time so as to reach the optimal value. As we show later in Section 4.3, this allows shuffle-once to converge faster than shuffle-always (between 2X-6X faster on the tasks we studied).

3.3 Parallelizing Gradient Computations

We now study how we can parallelize the IGD aggregate computation to achieve performance speed-ups on a single-node multicore system. We explain two mechanisms for achieving this parallelism – one based on standard UDA features, and another based on shared-memory features. We emphasize that both features are available in almost all RDBMSes.

Pure UDA Version. The UDA infrastructure offered by most RDBMSes (including the commercial DBMS A and DBMS B) include an built-in mechanism for 'shared-nothing' parallelism. The RDBMS requires that the developer provide a function `merge(state, state)`, along with the 3 functions discussed in Section 3.1. The `merge` function specifies how two aggregation contexts that were computed independently in parallel can be combined. For example, for an `AVG` query, two individual averages with sufficient statistics (total count) can be combined to obtain a new average. Generally, only aggregates that are *commutative* and *algebraic* can be parallelized in the above manner [8]. Although the IGD is not commutative, we observe that it is essentially commutative, in that it eventually converges to the optimal value *regardless* the data order (Section 3.2). And although the IGD is not algebraic, recent results from the machine learning community suggest that one can achieve rapid convergence by averaging models (trained on different portions of the data) [53]. Thus, the IGD is essentially algebraic as well. In turn, this implies that we can use the parallel UDA approach to achieve near-linear speed-ups on the IGD aggregate computations.

Shared-Memory UDA. Shared-memory management is provided by most RDBMSes [6], and it enables us to implement the IGD aggregate completely in the user space with no changes needed to the RDBMS code. This allows us to preserve the 3-function abstraction from Section 3.1, and also reuse most of the code from the UDA-based implementation. The model to be learned is maintained in shared memory and is concurrently updated by parallel threads operating on different segments of the data. Concurrent updates suggest that we need locking on the shared model. Nevertheless, recent results from the machine learning community show that IGD can be parallelized in a shared-memory environment with no locking at all [37]. We adopt this technique into Bismarck. Light-weight locking schemes often have stronger theoretical properties for convergence, and so we consider one such scheme called Atomic Incremental Gradient (AIG) that uses only *CompareAndExchange* instructions to effectively perform per-component locking [37].

As shown later in Section 4, we empirically observe that the model-averaging approach (pure UDA) has a worse convergence rate than the shared-memory UDA, and so worse overall performance. This led us to consider the shared-memory UDA for Bismarck.

3.4 Avoiding Shuffling Overhead

From the CA-TX example in Section 3.2, we saw that bad data orderings can impact convergence, and that shuffling once suffices in some instances to achieve good convergence rate. However, shuffling even once could be expensive for very large datasets. We verified this on a scalability dataset, and it did not finish shuffling even in one day. Thus, we investigate if it is possible to achieve good convergence rate even on bad data orderings without any shuffling. A classical technique to cope with this situation is to *subsample* the data using *reservoir sampling* (in fact, some vendors do implement subsampling); in this technique, given an in-memory buffer size B, we can obtain a without-replacement sample of size B in just one pass over the dataset, without shuffling the dataset [46]. The main idea of reservoir sampling is straightforward: suppose that our reservoir (array) can hold m items and our goal is to sample from N ($\geq m$) items. Read the first m items and fill the reservoir. Then, when we read the kth additional item ($m + k$ overall), we randomly select an integer s in $[0, m + k)$. If $s < m$, then we put the item at slot s; otherwise we drop the item.

Empirically, we observe that the subsampling may have slow convergence. Our intuition is that the reservoir discards valuable data items that could be used to help the model converge faster. To address this issue, we propose a simple scheme that we call *multiplexed* reservoir sampling (MRS), which combines the reservoir sampling idea with the concurrent model updates idea from Section 3.3.

Multiplexed Reservoir Sampling. The multiplexed reservoir sampling (MRS) idea is to combine, or *multiplex*, gradient steps over both the reservoir sample and the data that is not put in the reservoir buffer. By using the reservoir sample, which is a valuable without-replacement sample, and the rest of the data in conjunction, our scheme can achieve faster convergence than subsampling.

As Figure 6 illustrates, in MRS, there are two threads that update the shared model concurrently, called the I/O Worker and the Memory Worker. The I/O Worker has two tasks: (1) it performs a standard gradient step (exactly as the previous code), and (2) it places tuples into a reservoir.

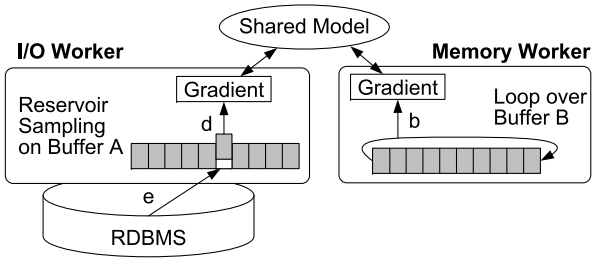


Figure 6: Multiplexed Reservoir Sampling (MRS): The I/O Worker reads example tuple e from the database, and uses buffer A to do reservoir sampling. The dropped example d is used for the gradient step, with updates to a shared model. The Memory Worker iterates over buffer B, and performs gradient steps on each example b in B concurrently.

Both of these functions are performed within the previously discussed UDA framework. The Memory Worker takes a buffer as input, and it loops over that buffer updating the model using the gradient rule. After the I/O Worker finishes one pass over the data, the buffers are swapped. That is, the I/O Worker begins filling the buffer that the Memory Worker is using, while the Memory Worker works on the buffer that has just been filled by the I/O Worker. The Memory Worker is signaled by polling a common integer indicating which buffer it should run over and whether it should continue running. In Section 4, we show that even with a buffer size that is an order of magnitude smaller than the dataset, MRS can achieve better convergence rates than both no-shuffling and subsampling.

4. EXPERIMENTS

We first show that our architecture, Bismarck, incurs little overhead, in terms of both development effort to add new analytics tasks, and runtime overhead inside an RDBMS. We then validate that Bismarck, implemented over two commercial RDBMSes and PostgreSQL, provides competitive or better performance than the native analytics tools offered by these RDBMSes on popular in-database analytics tasks. Finally, we evaluate how the generic optimizations that we described in Section 3 impact Bismarck's performance.

Dataset	Dimension	# Examples	Size
Forest	54	581k	77M
DBLife	41k	16k	2.7M
MovieLens	6k x 4k	1M	24M
CoNLL	7.4M	9K	20M
Classify300M	50	300M	135G
Matrix5B	706k x 706k	5B	190G
DBLP	600M	2.3M	7.2G

Table 1: Dataset Statistics. DBLife, CoNLL and DBLP are in sparse-vector format. MovieLens and Matrix5B are in sparse-matrix format.

Tasks and Datasets. We study 4 popular analytics tasks: Logistic Regression (LR), Support Vector Machine classification (SVM), Low-rank Matrix Factorization (LMF) and Conditional Random Fields labeling (CRF). We use 4 publicly available real-world datasets. For LR and SVM, we use two datasets – one dense (Forest, a standard benchmark dataset from the UCI repository) and one sparse (DBLife, which classifies papers by research areas). We binarized these datasets for the standard binary LR and SVM tasks. For LMF, we use MovieLens, which is a movie recommendation dataset, and for CRF, we use the CoNLL dataset, which is for text chunking. We also perform a scalability study with much larger datasets – two synthetic datasets Classify300M (for LR and SVM) and Matrix5B (for LMF), as well as DBLP (another real-world dataset) for CRF. The relevant statistics for all datasets are presented in Table 1.

Experimental Setup. All experiments are run on an identical configuration: a dual Xeon X5650 CPUs (6 cores each x 2 hyper-threading) machine with 128GB of RAM and a 1TB dedicated disk. The kernel is Linux 2.6.32-131. Each reported runtime is the average of three warm-cache runs. Completion time for gradient schemes here means achieving 0.1% tolerance in the objective function value, unless specified otherwise.

4.1 Overhead of Our Architecture

We first validate that Bismarck incurs little development overhead to add new analytics tasks. We then empirically verify that the runtime overhead of the tasks in Bismarck is low compared to a strawman aggregate.

Development Overhead. We implemented the 4 analytics tasks in Bismarck over three RDBMSes (PostgreSQL, commercial DBMS A and DBMS B). Bismarck enables rapid addition of a new analytics task since a large fraction of the code is shared across all the techniques implemented (on a given RDBMS). For example, starting with an end-to-end implementation of LR in Bismarck (in C, over PostgreSQL), we need to modify fewer than two dozen lines of code in order to add the SVM module.[4] Similarly, we can easily add in a more sophisticated task like LMF with only five dozen new lines of code. We believe that this is possible because our unified architecture based on IGD abstracts out the logic of the various tasks into a small number of generic functions. This is in contrast to existing systems, where there is usually a dedicated code stack for each task.

Runtime Overhead. We next verify that the tasks implemented in Bismarck have low runtime overhead. To do this, we compared our implementation to a strawman aggregate that sees the same data, but computes no values. We call this a NULL aggregate. We run three tasks – LR, SVM and LMF in Bismarck over all the 3 RDBMSes, using both the pure UDA infrastructure (shared-nothing) and the shared-memory variant described in Section 3. We compare the single-iteration runtime of each task against the NULL aggregate for both implementations of Bismarck over the same datasets. The results are presented in Tables 2 and 3.

We see that the overhead compared to the `NULL` aggregate can be as low as 4.6%, and is rarely more than 2X runtime for simple tasks like LR and SVM. The overhead is higher for the more computation-intensive task LMF, but is still less than 2.5X runtime of the `NULL` aggregate. We also see

[4]Both our code and the data used in our experiments are available at: http://research.cs.wisc.edu/hazy/victor/bismarck-download/

PostgreSQL				DBMS A				DBMS B (8 segments)			
Dataset (NULL time)	Tasks	Run--time	Over--head	Dataset (NULL time)	Tasks	Run--time	Over--head	Dataset (NULL time)	Tasks	Run--time	Over--head
Forest (0.3s)	LR	0.57s	90%	Forest (20.9s)	LR	24.1s	15.3%	Forest (0.14s)	LR	0.17s	21.4%
	SVM	0.56s	83.3%		SVM	22.0s	5.26%		SVM	0.16s	14.3%
DBLife (0.012s)	LR	0.035s	192%	DBLife (0.59)	LR	1.1s	86.4%	DBLife (0.085s)	LR	0.1	17.6%
	SVM	0.03s	150%		SVM	0.8s	35.6%		SVM	0.096s	12.9%
MovieLens (0.25s)	LMF	0.86s	244%	MovieLens (35.4s)	LMF	45.8s	29.4%	MovieLens (0.16s)	LMF	0.32s	100%

Table 2: *Pure UDA* implementation overheads: single-iteration runtime of each task implemented in Bismarck against the strawman NULL aggregate. The parallel database DBMS B was run with 8 segments.

PostgreSQL				DBMS A				DBMS B (8 segments)			
Dataset (NULL time)	Tasks	Run--time	Over--head	Dataset (NULL time)	Tasks	Run--time	Over--head	Dataset (NULL time)	Tasks	Run--time	Over--head
Forest (0.3s)	LR	0.56s	86.7%	Forest (3.3s)	LR	5.1s	54.5%	Forest (0.1s)	LR	0.25s	150%
	SVM	0.55s	83.3%		SVM	4.0s	21.2%		SVM	0.21s	110%
DBLife (0.012s)	LR	0.017s	41.7%	DBLife (0.11s)	LR	0.2s	81.8%	DBLife (0.043s)	LR	0.045s	4.6%
	SVM	0.016s	33.3%		SVM	0.3s	172%		SVM	0.045s	4.6%
MovieLens (0.29s)	LMF	0.85s	193%	MovieLens (5.1s)	LMF	10.3s	102%	MovieLens (0.1s)	LMF	0.26s	160%

Table 3: *Shared-memory UDA* implementation overheads: single-iteration runtime of each task implemented in Bismarck against the strawman NULL aggregate. The parallel database DBMS B was run with 8 segments.

Dataset	Task	PostgreSQL		DBMS A		DBMS B (8 segments)	
		Bismarck	MADlib	Bismarck	Native	Bismarck	Native
Forest (Dense)	LR	8.0	43.5	40.2	489.0	3.7	17.0
	SVM	7.5	140.2	32.7	66.7	3.3	19.2
DBLife (Sparse)	LR	0.8	N/A	9.8	20.6	2.3	N/A
	SVM	1.2	N/A	11.6	4.8	4.1	N/A
MovieLens	LMF	36.0	29325.7	394.7	N/A	11.9	17431.3

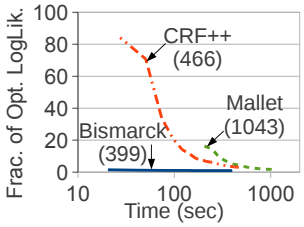


Figure 7: Benchmark Comparison: (A) Runtimes (in sec) for convergence (0.1% tolerance) or completion on 3 in-RDBMS analytics tasks. We compare Bismarck implemented over each RDBMS against the analytics tool native to that RDBMS. N/A means the task is not supported on that RDBMS' native tool (B) For the CRF task, we compare Bismarck (over PostgreSQL) against custom tools by plotting the objective function value against time. Completion times (in sec) are shown in parentheses.

that the shared-memory variant is several times faster than the UDA implementation over DBMS A, since DBMS A has extra overheads (e.g., model passing, serializations, etc.) to run the pure UDA. It was this observation that prompted us to use the shared-memory UDA to implement Bismarck even for a single-thread RDBMS.

4.2 Benchmark Comparison

We now validate that Bismarck implemented over two commercial RDBMSes and PostgreSQL provides competitive or better performance than the native analytics tools offered by these RDBMSes on three existing in-database analytics tasks – LR, SVM and LMF. For the comparison, we use the shared-memory UDA implementation of Bismarck along with the shuffle-once approach described in Section 3.2. For the parallel version of Bismarck, we use the no-lock shared-memory parallelism described in Section 3.3.

Competitor Analytics Tools. We compare Bismarck against three existing in-RDBMS tools – MADlib (an open-source collection of in-RDBMS statistical techniques [17]), which is run over PostgreSQL (single-threaded), and the native analytics tools provided by the two commercial engines – DBMS A (single-threaded), and the parallel DBMS B (with 8 segments). We tuned the parameters for each tool, including Bismarck, on each task based on an extensive search in the parameter space. The data was preprocessed appropriately for all tools. Some of the tasks we study are not currently supported in the above tools. In particular, the CRF task is not available in any of the existing in-RDBMS analytics tools we considered, and so we compare Bismarck (over PostgreSQL) against the custom tools CRF++ [28] and Mallet [34].

Existing In-RDBMS Analytics Tasks. We first compare the end-to-end runtimes of the various tools on LR, SVM and LMF. The results are summarized in Figure 7 (A). Overall, we see that Bismarck implemented over each RDBMS has competitive or faster performance on all these tasks against the native tool of the respective RDBMS. On sim-

ple tasks like LR and SVM, we see that Bismarck is often several times faster than existing tools. That is, on the dense LR task, Bismarck is about 12X faster than DBMS A's tool, and about 5X faster than MADlib over both PostgreSQL and the native tool in DBMS B. In some cases, e.g., DBMS A for sparse SVM, Bismarck is slightly slower due to the function call overheads in DBMS A. On a more complex task like LMF, we see that Bismarck is about 3 orders-of-magnitude faster than MADlib and DBMS B's native tool. This validates that Bismarck is able to efficiently handle several in-RDBMS analytics tasks, while offering a unified architecture. We also verified that all the tools compared achieved similar training quality on a given task and dataset (recall that IGD converges to the optimal objective value on convex programs), but do not present details here due to space constraints.

To understand why Bismarck performs faster, we looked into the MADlib source code. While the reasons vary across tasks, Bismarck is faster generally because IGD has lower time complexity than the algorithms in MADlib. IGD, across all tasks, is linear in the number of examples (fixing the dimension) and linear in the dimension of the model (fixing the number of examples). But the algorithms in MADlib for LR, for instance, are super-linear in the dimension, while that for LMF is super-linear in the number of examples.

To get a sense of the performance compared to other tools, a comparison with the popular in-memory tool Weka shows that Bismarck (over PostgreSQL) is faster on all these tasks – from 4X faster on dense LR to over 4000X faster on dense SVM. We also validated that our runtimes on SVM are within a factor of 3X to the special-purpose SVM in-memory tool, SVMPerf. This is not surprising as SVMPerf is highly optimized for the SVM computation, but presents an avenue for future work.

Next Generation Tasks. Existing in-RDBMS analytics tools do not support emerging advanced analytics tasks like CRF. But Bismarck is able to efficiently support even such next generation tasks within the same architecture. To validate this, we plot the convergence over time for Bismarck (over PostgreSQL) against in-memory tools. The results are shown in Figure 7(B). We see that Bismarck is able to achieve similar convergence, and runtime as the hand-coded and optimized in-memory tools, even though Bismarck is a more generic in-RDBMS tool.

Task	Bismarck PostgreSQL	DBMS A (Native)	DBMS B (Native)	Others (In-mem.)
LR	√	√	√	X
SVM	√	√	X	X
LMF	√	N/A	X	X
CRF	√	N/A	N/A	X

Table 4: Scalability : √ means the task completes, and X means that the approach either crashes or takes longer than 48 hours. N/A means the task is not supported. The in-memory tools (Weka, SVMPerf, CRF++, Mallet) all either crash or take too long.

Scalability. We now study the scalability of the various tools to much larger datasets (Classify300M, Matrix5B and DBLP). Since Bismarck is not tied to any RDBMS, we run it over PostgreSQL for this study. We compare against the native analytics tools of both commercial engines, DBMS A and DBMS B, as well as the task-specific in-memory tools mentioned before. The results are summarized in Table 4.2. We see that almost all of the in-RDBMS tools scale on the simple tasks LR and SVM (less than an hour per epoch for Bismarck), except DBMS B on SVM, which did not terminate even after 48 hours. Again, on the more complex tasks LMF and CRF, only Bismarck scales to the large datasets. We also tried several custom in-memory tools – all crashed either due to insufficient memory (Weka, SVMPerf, CRF++) or did not terminate even after 48 hours (Mallet).

4.3 Impact of Data Ordering

We now empirically verify how the order the data is stored affects the performance of our IGD schemes. We first study the objective function value against epochs for data being shuffled before each epoch (ShuffleAlways). We repeat the study for data seen in clustered order (Clustered), without any shuffling. Finally, we shuffle the data only once, before the first epoch (ShuffleOnce). We present the results for the LR task on DBLife in Figure 8. We observed similar results on other datasets and tasks, but skip them here due to space constraints.

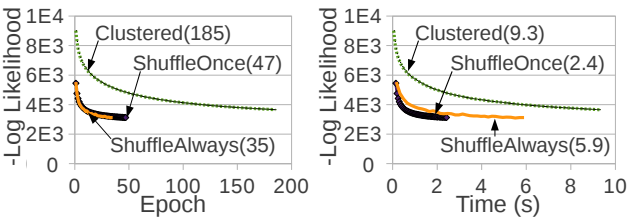


Figure 8: Impact of Data Ordering on Sparse LR over DBLife: (A) Objective value over epochs, till convergence. The number of epochs for convergence are shown in parentheses. (B) Objective value over time, till convergence. The time to converge (in sec) are shown in parentheses.

Figure 8(A) shows that ShuffleAlways converges in the fewest epochs, as is expected for IGD. Clustered yields the poorest convergence rate, as explained in Section 3.2. In fact, Clustered takes over 1000 epochs to reach the same objective value as ShuffleAlways. However, we see that ShuffleOnce achieves very similar convergence rate to ShuffleAlways, and reaches the same objective value as ShuffleAlways in 12 extra epochs. Figure 8(B) shows why the extra epochs are acceptable – ShuffleAlways takes several times longer to finish than ShuffleOnce. This is because the shuffling overhead is significantly high. In fact, for simple tasks like LR, shuffling dominates the runtime – e.g., for LR on DBLife, shuffling takes nearly 5X the time for gradient computation per epoch. Even on more complex tasks, the overhead is significant, e.g., it is 3X for LMF on MovieLens. By avoiding this overhead, ShuffleOnce finishes much faster than ShuffleAlways, while still achieving the same quality.

4.4 Parallelizing IGD in an RDBMS

We now verify that both the parallelism schemes (pure UDA and shared-memory UDA) are able to achieve near-linear speed-ups but the pure UDA has a worse convergence

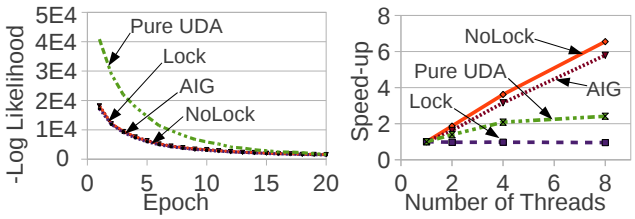


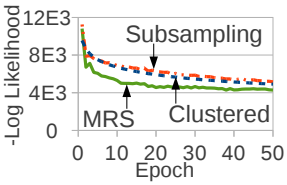
Figure 9: Parallelizing IGD: (A) Plot of objective value over epochs for the pure UDA version and the shared-memory UDA variants (Lock, AIG, NoLock) for CRF over CoNLL on 8 threads (segments). (B) Speed-up of the per-epoch gradient computation times against the number of threads. The per-epoch time of the single-threaded run is 20.6s.

rate than the shared-memory UDA. We first study the objective value over epochs for both the implementations. We use the three concurrency schemes for the shared-memory UDA – lock the model (Lock), AIG, and no locking (NoLock). We present the results for CRF on CoNLL in Figure 9(A) (similar results on other tasks skipped here for brevity).

Figure 9(A) shows that the pure UDA implementation has poorer convergence rate compared to the shared-memory UDA with Lock, since the model averaging in the former yields poorer quality [52]. The figure also shows that AIG and NoLock have similar convergence rate to the Lock approach. This is in line with recent results from the machine learning literature [37]. By adopting the NoLock shared-memory UDA parallelism into Bismarck, we achieve significant speed-ups in a generic way across all the analytics tasks we handle. Figure 9(B) shows the speed-ups (over a single-threaded run) achieved by the four parallelism schemes in DBMS B. As expected, the Lock approach has no speed-up, while the speed-up of the pure UDA approach is sub-optimal due to model passing overheads. NoLock and AIG achieve linear speed-ups, with NoLock having the highest speed-ups.

4.5 Multiplexed Reservoir Sampling

We verify that our Multiplexed Reservoir Sampling (MRS) scheme has faster convergence rate compared to both Subsampling and operating over clustered data (Clustered).



B	Sub-Sampling	MRS
800	2.50 (48)	0.60 (10)
1600	1.37 (26)	0.36 (6)
3200	0.69 (13)	0.12 (2)

Figure 10: Multiplexed Reservoir Sampling: (A) Objective value against epochs for LR on DBLife. The buffer size for Subsampling and MRS is 1600 tuples (10% of the dataset). (B) Runtime (in sec) to reach 2X the optimal objective value for different buffer sizes, B. The numbers in parentheses indicate the respective number of epochs. The same values for Clustered are 1.03s (19).

Figure 10(A) plots the objective value against epochs for the three schemes. For Subsampling and MRS, we choose a buffer size that is about 10% the dataset size (for LR on DBLife). We see from the figure that MRS has faster convergence rate than both Subsampling and Clustered, and reaches an objective value that is 20% lower than both. Figure 10(B) shows the sensitivity to the buffer size for the Subsampling and MRS schemes. We see that the runtime to reach 2X of the optimal objective value is lower for MRS. This is as expected since MRS has faster convergence rate than Subsampling. Finally, we verify that Bismarck with the MRS scheme provides better performance than existing in-RDBMS tools on large datasets (that do not fit in available RAM). For a simple task like LR on the Classify300M dataset over PostgreSQL, with a buffer that is just 1% of the dataset size, Bismarck with the MRS scheme achieves the same objective value as MADlib in 45 minutes, while MADlib takes over 3 hours. On a more complex task like LMF on the Matrix5B dataset, Bismarck with MRS scheme finishes in a few hours, while MADlib did not terminate even after one week.

5. CONCLUSIONS AND FUTURE WORK

We present Bismarck, a novel architecture that takes a step towards unifying in-RDBMS analytics. Using insights from the mathematical programming literature, Bismarck provides a single systems-level abstraction to implement a large class of existing and next-generation analytics techniques. In providing a unified architecture, we argue that Bismarck may reduce the development overhead for introducing and maintaining sophisticated analytics code in an RDBMS. Bismarck also achieves high performance on these techniques by effectively utilizing standard features available inside every RDBMS. We implemented Bismarck over two commercial RDBMSes and PostgreSQL, and verified that Bismarck achieves competitive, and often superior, performance than the state-of-the-art analytics tools natively offered by these RDBMSes.

While Bismarck can handle many analytics techniques in the current framework, it is interesting future work to integrate more sophisticated models, e.g., simulation models, into our architecture. Another direction is to handle large-scale combinatorial optimization problems inside the RDBMS, including tasks like linear programming and fundamental NP-hard problems like MAX-CUT.

One area to improve Bismarck is to match the performance of some specialized tools for tasks like support vector machines by using more optimizations, e.g. model or feature compression. There are also possibilities to improve performance by modifying the DBMS engine, e.g., exploiting better mechanisms for model passing and storage, concurrency control, etc. Another direction is to examine more fully how to utilize features that are available in parallel RDBMSes.

6. ACKNOWLEDGMENTS

This research has been supported by the ONR grant N00014-12-1-0041, the NSF CAREER award IIS-1054009, and gifts from EMC Greenplum and Oracle to Christopher Ré, and by the ONR grant N00014-11-1-0723 to Benjamin Recht. We also thank Joseph Hellerstein, and the analytics teams from EMC Greenplum and Oracle for invaluable discussions.

7. REFERENCES

[1] Microsoft SQL Server 2008 R2 Data Mining.
[2] Microsoft SQL Server Books Online.
[3] Oracle Data Cartridge Developer's Guide 11g.
[4] Oracle Data Mining.

[5] PostgreSQL 9.0 Documentation.
[6] Shared Memory and LWLocks in PostgreSQL.
[7] Vowpal Wabbit. http://hunch.net/~vw/.
[8] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
[9] R. Agrawal and R. Srikant. Fast Algorithms for Mining Association Rules in Large Databases. In *VLDB*, pages 487–499, 1994.
[10] K. M. Anstreicher and L. A. Wolsey. Two "Well-known" Properties of Subgradient Optimization. *Math. Program.*, 120(1):213–220, 2009.
[11] D. P. Bertsekas. A Hybrid Incremental Gradient Method for Least Squares. *SIAM Journal on Optimization*, 7, 1997.
[12] D. P. Bertsekas. *Nonlinear Programming*. Athena Scientific, Belmont, MA, 2nd edition, 1999.
[13] D. P. Bertsekas. Incremental Gradient, Subgradient, and Proximal Methods for Convex Optimization: A Survey. Technical report, Laboratory for Information and Decision Systems, 2010.
[14] L. Bottou and O. Bousquet. The Tradeoffs of Large Scale Learning. In *NIPS*, 2007.
[15] L. Bottou and Y. LeCun. Large Scale Online Learning. In *NIPS*, 2003.
[16] S. Boyd and L. Vandenberghe. *Convex Optimization*. Cambridge University Press, New York, NY, USA, 2004.
[17] J. Cohen, B. Dolan, M. Dunlap, J. M. Hellerstein, and C. Welton. MAD Skills: New Analysis Practices for Big Data. *PVLDB*, 2(2):1481–1492, 2009.
[18] O. Dekel, R. Gilad-Bachrach, O. Shamir, and L. Xiao. Optimal Distributed Online Prediction. In *ICML*, pages 713–720, 2011.
[19] A. Deshpande and S. Madden. MauveDB: Supporting Model-based User Views in Database Systems. In *SIGMOD*, pages 73–84, 2006.
[20] J. Duchi, A. Agarwal, and M. J. Wainwright. Distributed Dual Averaging in Networks. In *NIPS*, 2010.
[21] EMC Greenplum. Personal Communication.
[22] X. Feng, A. Kumar, B. Recht, and C. Ré. Towards a Unified Architecture for in-RDBMS Analytics. *CoRR*, abs/1203.2574, 2012.
[23] R. Gemulla, E. Nijkamp, P. J. Haas, and Y. Sismanis. Large-scale Matrix Factorization with Distributed Stochastic Gradient Descent. In *KDD*, pages 69–77, 2011.
[24] M. X. Goemans and D. P. Williamson. Approximation Algorithms for MAX-3-CUT and Other Problems via Complex Semidefinite Programming. *J. Comput. Syst. Sci.*, 68(2), 2004.
[25] R. Gupta and S. Sarawagi. Creating Probabilistic Databases from Information Extraction Models. In *VLDB*, pages 965–976, 2006.
[26] T. Hastie, R. Tibshirani, and J. H. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. New York: Springer-Verlag, 2001.
[27] 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 687–698, 2008.
[28] T. Kudo. CRF++: Yet Another CRF Toolkit.
[29] J. Langford, L. Li, and T. Zhang. Sparse Online Learning via Truncated Gradient. *JMLR*, 10:777–801, 2009.
[30] Z. Luo. On the Convergence of the LMS Algorithm with Adaptive Learning Rate for Linear Feedforward Networks. *Neural Computation*, 3(2):226–245, 1991.
[31] Z. Q. Luo and P. Tseng. Analysis of an Approximate Gradient Projection Method with Applications to the Backpropagation Algorithm. *Optimization Methods and Software*, 4, 1994.
[32] O. L. Mangasarian. Linear and Nonlinear Separation of Patterns by Linear Programming. *Operations Research*, 13, 1965.
[33] O. L. Mangasarian and M. V. Solodov. Serial and Parallel Backpropagation Convergence via Nonmonotone Perturbed Minimization. *Optimization Methods and Software*, 4, 1994.
[34] A. McCallum. MALLET: A Machine Learning for Language Toolkit, 2002.
[35] B. L. Milenova, J. Yarmus, and M. M. Campos. SVM in Oracle Database 10g: Removing the Barriers to Widespread Adoption of Support Vector Machines. In *VLDB*, pages 1152–1163, 2005.
[36] A. Nemirovski, A. Juditsky, G. Lan, and A. Shapiro. Robust Stochastic Approximation Approach to Stochastic Programming. *SIAM Journal on Optimization*, 19(4), 2009.
[37] F. Niu, B. Recht, C. Ré, and S. Wright. Hogwild: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent. In *NIPS*, 2011.
[38] Oracle Advanced Analytics, Oracle R Enterprise Group. Personal Communication.
[39] P. Raghavendra. Optimal Algorithms and Inapproximability Results for Every CSP? In *STOC*, pages 245–254, 2008.
[40] H. Robbins and S. Monro. A Stochastic Approximation Method. *Ann. Math. Statistics*, 22(3):400–407, 1951.
[41] R. T. Rockafellar. Monotone Operators and the Proximal Point Algorithm. *SIAM J. on Control and Optimization*, 14(5), 1976.
[42] R. T. Rockafellar. *Convex Analysis (Princeton Landmarks in Mathematics and Physics)*. Princeton University Press, 1996.
[43] P. Sen, A. Deshpande, and L. Getoor. Exploiting Shared Correlations in Probabilistic Databases. *PVLDB*, 1(1):809–820, 2008.
[44] R. Tibshirani. Regression Shrinkage and Selection via the Lasso. *Journal of the Royal Statistical Society, B*, 58(1), 1996.
[45] P. Tseng. An Incremental Gradient(-Projection) Method with Momentum Term and Adaptive Stepsize Rule. *SIAM Joural on Optimization*, 8(2), 1998.
[46] J. S. Vitter. Random Sampling with a Reservoir. *ACM Trans. Math. Softw.*, 11(1):37–57, 1985.
[47] G. Wahba, C. Gu, Y. Wang, and R. Chappell. Soft Classification, a.k.a. Risk Estimation, via Penalized Log Likelihood and Smoothing Spline Analysis of Variance. In *The Mathematics of Generalization.*, Santa Fe Institute Studies in the Sciences of Complexity. Addison-Wesley, 1995.
[48] H. M. Wallach. Conditional Random Fields: An Introduction. Technical report, Dept. of CIS, Univ. of Pennsylvania, 2004.
[49] D. Z. Wang, M. J. Franklin, M. N. Garofalakis, and J. M. Hellerstein. Querying Probabilistic Information Extraction. *PVLDB*, 3(1):1057–1067, 2010.
[50] D. Z. Wang, E. Michelakis, M. Garofalakis, and J. M. Hellerstein. BayesStore: Managing Large, Uncertain Data Repositories with Probabilistic Graphical Models. *PVLDB*, 1(1):340–351, 2008.
[51] M. Wick, A. McCallum, and G. Miklau. Scalable Probabilistic Databases with Factor Graphs and MCMC. *PVLDB*, 3(1):794–804, 2010.
[52] Z. A. Zhu, W. Chen, G. Wang, C. Zhu, and Z. Chen. P-packSVM: Parallel Primal grAdient desCent Kernel SVM. In *ICDM*, pages 677–686, 2009.
[53] M. Zinkevich, M. Weimer, A. Smola, and L. Li. Parallelized Stochastic Gradient Descent. In *NIPS*, 2010.

APPENDIX

A. PROXIMAL POINT METHODS

To handle regularization and constraints, we need an additional concept called *proximal point methods*. These do not change the data access patterns, but do enable us to handle constraints. We state the complete step rule including a projection that allows us to handle constraints:

$$w^{(k+1)} = \Pi_{\alpha P}\left(w^{(k)} - \alpha_k \nabla f_{\eta(k)}(w^{(k)})\right) \tag{3}$$

Where the function $\Pi_{\alpha P}$ is called a proximal point operator and is defined by the expression:

$$\Pi_{\alpha P}(x) = \arg\min_w \quad \tfrac{1}{2}\|x - w\|_2^2 + \alpha P(w)$$

In the case where P is the indicator function of a set C, $\Pi_{\alpha P}$ is simply the Euclidean projection onto C [41]. Thus, these constraints can be used to ensure that the model stays in some convex set of constraints. An example proximal-point operator ensures that the model has unit Euclidean norm by projecting the model on to the the unit ball. $P(w)$ might also be a regularization penalty such as total-variation or negative entropy. These are very commonly used in statistics to improve the generalization of the model or to take advantage of properties that are known about the model to reduce the number of needed measurements.