DimmWitted: A Study of Main-Memory Statistical Analytics

Ce Zhang[†‡] Christopher Ré[†]
[†]Stanford University
[‡]University of Wisconsin-Madison
{czhang, chrismre}@cs.stanford.edu

ABSTRACT

We perform the first study of the tradeoff space of access methods and replication to support statistical analytics using first-order methods executed in the main memory of a Non-Uniform Memory Access (NUMA) machine. Statistical analytics systems differ from conventional SQL-analytics in the amount and types of memory incoherence that they can tolerate. Our goal is to understand tradeoffs in accessing the data in row- or column-order and at what granularity one should share the model and data for a statistical task. We study this new tradeoff space and discover that there are tradeoffs between hardware and statistical efficiency. We argue that our tradeoff study may provide valuable information for designers of analytics engines: for each system we consider, our prototype engine can run at least one popular task at least 100× faster. We conduct our study across five architectures using popular models, including SVMs, logistic regression, Gibbs sampling, and neural networks.

1. INTRODUCTION

Statistical data analytics is one of the hottest topics in data-management research and practice. Today, even small organizations have access to machines with large main memories (via Amazon's EC2) or for purchase at $5/GB. As a result, there has been a flurry of activity to support main-memory analytics in both industry (Google Brain, Impala, and Pivotal) and research (GraphLab, and MLlib). Each of these systems picks one design point in a larger tradeoff space. The goal of this paper is to define and explore this space. We find that today's research and industrial systems under-utilize modern commodity hardware for analytics—sometimes by two orders of magnitude. We hope that our study identifies some useful design points for the next generation of such main-memory analytics systems.

Throughout, we use the term *statistical analytics* to refer to those tasks that can be solved by *first-order methods*–a class of iterative algorithms that use gradient information; these methods are the core algorithm in systems such as MLlib, GraphLab, and Google Brain. Our study examines analytics on commodity multi-socket, multi-core, non-uniform memory access (NUMA) machines, which are the de facto standard machine configuration and thus a natural target for an in-depth study. Moreover, our experience with several enterprise companies suggests that, after appropriate preprocessing, a large class of enterprise analytics problems fit into the main memory of a single, modern machine. While this architecture has been recently studied for traditional SQL-analytics systems [9], it has not been studied for *statistical* analytics systems.

Statistical analytics systems are different from traditional SQL-analytics systems. In comparison to traditional SQL-analytics, the underlying methods are intrinsically robust to error. On the other hand, traditional statistical theory does not consider which operations can be efficiently executed. This leads to a fundamental tradeoff between *statistical efficiency* (how many steps are needed until convergence to a given tolerance) and *hardware efficiency* (how efficiently those steps can be carried out).

To describe such tradeoffs more precisely, we describe the setup of the analytics tasks that we consider in this paper. The input data is a matrix in $\mathbb{R}^{N\times d}$ and the goal is to find a vector $x \in \mathbb{R}^d$ that minimizes some (convex) loss function, say the logistic loss or the hinge loss (SVM). Typically, one makes several complete passes over the data while updating the model; we call each such pass an *epoch*. There may be some communication at the end of the epoch, e.g., in bulk-synchronous parallel systems such as Spark. We identify three tradeoffs that have not been explored in the literature: (1) *access methods for the data*, (2) *model replication*, and (3) *data replication*. Current systems have picked one point in this space; we explain each space and discover points that have not been previously considered. Using these new points, we can perform 100× faster than previously explored points in the tradeoff space for several popular tasks.

Access Methods. Analytics systems access (and store) data in either row-major or column-major order. For example, systems that use *stochastic gradient descent methods* (SGD) access the data row-wise; examples include MADlib [13] in Impala and Pivotal, Google Brain [18], and MLlib in Spark [33]; and *stochastic coordinate descent methods* (SCD) access the data column-wise; examples include GraphLab [21], Shogun [32], and Thetis [34]. These methods have essentially identical statistical efficiency, but their wall-clock performance can be radically different due to hardware effi-

Proceedings of the VLDB Endowment, Vol. 7, No. 12

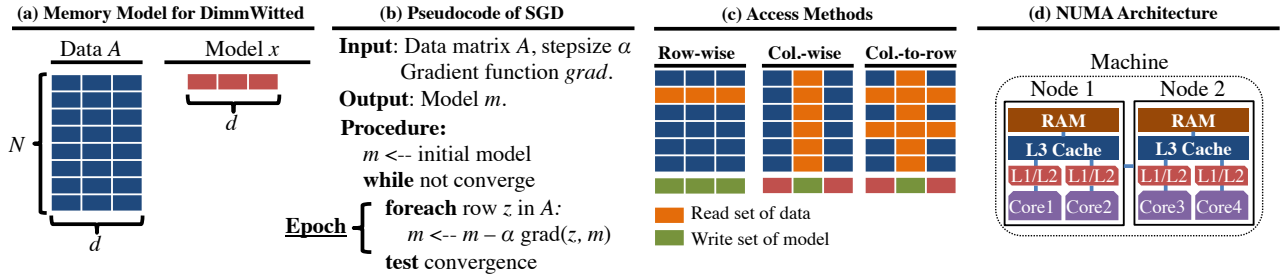

Figure 1: Illustration of (a) DimmWitted's Memory Model, (b) Pseudocode for SGD, (c) Different Statistical Methods in DimmWitted and Their Access Patterns, and (d) NUMA Architecture.

ciency. However, this tradeoff has not been systematically studied. To study this tradeoff, we introduce a storage abstraction that captures the access patterns of popular statistical analytics tasks and a prototype called DimmWitted. In particular, we identify three access methods that are used in popular analytics tasks, including standard supervised machine learning models such as SVMs, logistic regression, and least squares; and more advanced methods such as neural networks and Gibbs sampling on factor graphs. For different access methods for the same problem, we find that the time to converge to a given loss can differ by up to 100×.

We also find that no access method dominates all others, so an engine designer may want to include both access methods. To show that it may be possible to support both methods in a single engine, we develop a simple cost model to choose among these access methods. We describe a simple cost model that selects a nearly optimal point in our data sets, models, and different machine configurations.

Data and Model Replication. We study two sets of tradeoffs: the level of granularity, and the mechanism by which mutable state and immutable data are shared in analytics tasks. We describe the tradeoffs we explore in both (1) mutable state sharing, which we informally call *model replication*, and (2) *data replication*.

(1) Model Replication. During execution, there is some state that the task mutates (typically an update to the model). We call this state, which may be shared among one or more processors, a *model replica*. We consider three different granularities at which to share model replicas:

- The PerCore approach treats a NUMA machine as a distributed system in which every core is treated as an individual machine, e.g., in bulk-synchronous models such as MLlib on Spark or event-driven systems such as GraphLab. These approaches are the classical shared-nothing and event-driven architectures, respectively. In PerCore, the part of the model that is updated by each core is only visible to that core until the end of an epoch. This method is efficient and scalable from a hardware perspective, but it is less statistically efficient, as there is only coarse-grained communication between cores.

- The PerMachine approach acts as if each processor has uniform access to memory. This approach is taken in Hogwild! and Google Downpour [10]. In this method, the hardware takes care of the coherence of the shared state. The PerMachine method is statistically efficient due to high communication rates, but it may cause contention in the hardware, which may lead to suboptimal running times.

- A natural hybrid is PerNode; this method uses the fact that PerCore communication through the last-level cache (LLC) is dramatically faster than communication through remote main memory. This method is novel; for some models, PerNode can be an order of magnitude faster.

Because model replicas are mutable, a key question is *how often should we synchronize model replicas?* We find that it is beneficial to synchronize the models as much as possible—so long as we do not impede throughput to data in main memory. A natural idea, then, is to use PerMachine sharing, in which the hardware is responsible for synchronizing the replicas. However, this decision can be suboptimal, as the cache-coherence protocol may stall a processor to preserve coherence, but this information may not be worth the cost of a stall from a statistical efficiency perspective. We find that the PerNode method, coupled with a simple technique to batch writes across sockets, can dramatically reduce communication and processor stalls. The PerNode method can result in an over 10× runtime improvement. This technique depends on the fact that we do not need to maintain the model consistently: we are effectively delaying some updates to reduce the total number of updates across sockets (which lead to processor stalls).

(2) Data Replication. The data for analytics is immutable, so there are no synchronization issues for data replication. The classical approach is to partition the data to take advantage of higher aggregate memory bandwidth. However, each partition may contain skewed data, which may slow convergence. Thus, an alternate approach is to replicate the data fully (say, per NUMA node). In this approach, each node accesses that node's data in a different order, which means that the replicas provide non-redundant statistical information; in turn, this reduces the variance of the estimates based on the data in each replicate. We find that for some tasks, fully replicating the data four ways can converge to the same loss almost 4× faster than the sharding strategy.

Summary of Contributions. We are the first to study the three tradeoffs listed above for main-memory statistical analytics systems. These tradeoffs are not intended to be an exhaustive set of optimizations, but they demonstrate our

main conceptual point: *treating NUMA machines as distributed systems or SMP is suboptimal for statistical analytics.* We design a storage manager, DimmWitted, that shows it is possible to exploit these ideas on real data sets. Finally, we evaluate our techniques on multiple real datasets, models, and architectures.

2. BACKGROUND

In this section, we describe the memory model for DimmWitted, which provides a unified memory model to implement popular analytics methods. Then, we recall some basic properties of modern NUMA architectures.

Data for Analytics. The data for an analytics task is a pair (A, x), which we call the data and the model, respectively. For concreteness, we consider a matrix $A \in \mathbb{R}^{N \times d}$. In machine learning parlance, each row is called an *example*. Thus, N is often the number of examples and d is often called the dimension of the model. There is also a model, typically a vector $x \in \mathbb{R}^d$. The distinction is that the data A is read-only, while the model vector, x, will be updated during execution. From the perspective of this paper, the important distinction we make is that data is an immutable matrix, while the model (or portions of it) are mutable data.

First-Order Methods for Analytic Algorithms. DimmWitted considers a class of popular algorithms called *first-order methods*. Such algorithms make several passes over the data; we refer to each such pass as an *epoch*. A popular example algorithm is stochastic gradient descent (SGD), which is widely used by web-companies, e.g., Google Brain [18] and VowPal Wabbit [1], and in enterprise systems such as Pivotal, Oracle, and Impala. Pseudocode for this method is shown in Figure 1(b). During each epoch, SGD reads a single example z; it uses the current value of the model and z to estimate the derivative; and it then updates the model vector with this estimate. It reads each example in this loop. After each epoch, these methods test convergence (usually by computing or estimating the norm of the gradient); this computation requires a scan over the complete dataset.

2.1 Memory Models for Analytics

We design DimmWitted's memory model to capture the trend in recent high-performance sampling and statistical methods. There are two aspects to this memory model: the *coherence level* and the *storage layout*.

Coherence Level. Classically, memory systems are coherent: reads and writes are executed atomically. For analytics systems, we say that a memory model is *coherent* if reads and writes of the entire model vector are atomic. That is, access to the model is enforced by a critical section. However, many modern analytics algorithms are designed for an *incoherent* memory model. The Hogwild! method showed that one can run such a method in parallel without locking but still provably converge. The Hogwild! memory model relies on the fact that writes of individual components are atomic, but it does not require that the entire vector be updated atomically. However, atomicity at the level of the cacheline is provided by essentially all modern processors. Empirically, these results allow one to forgo costly locking (and coherence) protocols. Similar algorithms have been proposed for other popular methods, including Gibbs sampling [15, 31], stochastic coordinate descent (SCD) [29, 32], and linear systems solvers [34]. This technique was applied by Dean et al. [10] to solve convex optimization problems with billions of elements in a model. This memory model is distinct from the classical, *fully coherent* database execution.

Algorithm	Access Method	Implementation
Stochastic Gradient Descent	Row-wise	MADlib, Spark, Hogwild!
Stochastic Coordinate Descent	Column-wise	GraphLab, Shogun, Thetis
	Column-to-row	

Figure 2: Algorithms and Their Access Methods.

The DimmWitted prototype allows us to specify that a region of memory is coherent or not. This region of memory may be shared by one or more processors. If the memory is only shared per thread, then we can simulate a shared-nothing execution. If the memory is shared per machine, we can simulate Hogwild!.

Access Methods. We identify three distinct access paths used by modern analytics systems, which we call row-wise, column-wise, and column-to-row. They are graphically illustrated in Figure 1(c). Our prototype supports all three access methods. All of our methods perform several epochs, that is, passes over the data. However, the algorithm may iterate over the data row-wise or column-wise.

- In *row-wise access*, the system scans each row of the table and applies a function that takes that row, applies a function to it, and then updates the model. This method may write to all components of the model. Popular methods that use this access method include stochastic gradient descent, gradient descent, and higher-order methods (such as l-BFGS).
- In *column-wise access*, the system scans each column j of the table. This method reads just the j component of the model. The write set of the method is typically a single component of the model. This method is used by stochastic coordinate descent.
- In *column-to-row access*, the system iterates conceptually over the columns. This method is typically applied to sparse matrices. When iterating on column j, it will read all rows in which column j is non-zero. This method also updates a single component of the model. This method is used by non-linear support vector machines in GraphLab and is the de facto approach for Gibbs sampling.

DimmWitted is free to iterate over rows or columns in essentially any order (although typically some randomness in the ordering is desired). Figure 2 classifies popular implementations by their access method.

2.2 Architecture of NUMA Machines

We briefly describe the architecture of a modern NUMA machine. As illustrated in Figure 1(d), a NUMA machine contains multiple NUMA nodes. Each node has multiple cores and processor caches, including the L3 cache. Each node is directly connected to a region of DRAM. NUMA

Name (abbrv.)	#Node	#Cores/ Node	RAM/ Node (GB)	CPU Clock (GHz)	LLC (MB)
local2 (l2)	2	6	32	2.6	12
local4 (l4)	4	10	64	2.0	24
local8 (l8)	8	8	128	2.6	24
ec2.1 (e1)	2	8	122	2.6	20
ec2.2 (e2)	2	8	30	2.6	20

Figure 3: Summary of Machines and Memory Bandwidth on local2 Tested with STREAM [6].

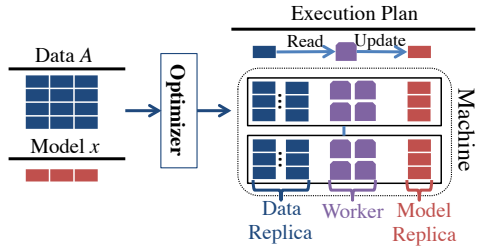

Figure 4: Illustration of DimmWitted's Engine.

nodes are connected to each other by buses on the main board; in our case, this connection is the Intel Quick Path Interconnects (QPIs), which has a bandwidth as high as 25.6GB/s.[1] To access DRAM regions of other NUMA nodes, data is transferred across NUMA nodes using the QPI. These NUMA architectures are cache coherent, and the coherency actions use the QPI. Figure 3 describes the configuration of each machine that we use in this paper. Machines controlled by us have names with the prefix "local"; the other machines are Amazon EC2 configurations.

3. THE DIMMWITTED ENGINE

We describe the tradeoff space that DimmWitted's optimizer considers, namely (1) access method selection, (2) model replication, and (3) data replication. To help understand the statistical-versus-hardware tradeoff space, we present some experimental results in a *Tradeoffs* paragraph within each subsection. We describe implementation details for DimmWitted in the full version of this paper.

3.1 System Overview

We describe analytics tasks in DimmWitted and the execution model of DimmWitted given an analytics task.

System Input. For each analytics task that we study, we assume that the user provides data $A \in \mathbb{R}^{N \times d}$ and an initial model that is a vector of length d. In addition, for each access method listed above, there is a function of an appropriate type that solves the same underlying model. For example, we provide both a row- and column-wise way of solving a support vector machine. Each method takes two arguments; the first is a pointer to a model.

- f_{row} captures the the row-wise access method, and its second argument is the index of a single row.
- f_{col} captures the column-wise access method, and its second argument is the index of a single column.
- f_{ctr} captures the column-to-row access method, and its second argument is a pair of one column index and a set of row indexes. These rows correspond to the non-zero entries in a data matrix for a single column.[2]

Each of the functions modifies the model to which they receive a pointer in place. However, in our study, f_{row} can modify the whole model, while f_{col} and f_{ctr} only modify a single variable of the model. We call the above tuple of functions a *model specification*. Note that a model specification contains either f_{col} or f_{ctr} but typically not both.

[1] www.intel.com/content/www/us/en/io/quickpath-technology/quick-path-interconnect-introduction-paper.html

Tradeoff	Strategies	Existing Systems
Access Methods	Row-wise	SP, HW
	Column-wise	GL
	Column-to-row	
Model Replication	Per Core	GL, SP
	Per Node	
	Per Machine	HW
Data Replication	Sharding	GL, SP, HW
	Full Replication	

Figure 5: A Summary of DimmWitted's Tradeoffs and Existing Systems (GraphLab (GL), Hogwild! (HW), Spark (SP)).

Execution. Given a model specification, our goal is to generate an execution plan. An execution plan, schematically illustrated in Figure 4, specifies three things for each CPU core in the machine: (1) a subset of the data matrix to operate on, (2) a replica of the model to update, and (3) the access method used to update the model. We call the set of replicas of data and models *locality groups*, as the replicas are described physically; i.e., they correspond to regions of memory that are local to particular NUMA nodes, and one or more workers may be mapped to each locality group. The data assigned to distinct locality groups may overlap. We use DimmWitted's engine to explore three tradeoffs:

(1) Access methods, in which we can select between either the row or column method to access the data.
(2) Model replication, in which we choose how to create and assign replicas of the model to each worker. When a worker needs to read or write the model, it will read or write the model replica that it is assigned.
(3) Data replication, in which we choose a subset of data tuples for each worker. The replicas may be overlapping, disjoint, or some combination.

Figure 5 summarizes the tradeoff space. In each section, we illustrate the tradeoff along two axes, namely (1) the *statistical efficiency*, i.e., the number of epochs it takes to converge, and (2) *hardware efficiency*, the time that each method takes to finish a single epoch.

3.2 Access Method Selection

In this section, we examine each access method: row-wise, column-wise, and column-to-row. We find that the execution time of an access method depends more on hardware efficiency than on statistical efficiency.

[2] Define $S(j) = \{i : a_{ij} \neq 0\}$. For a column j, the input to f_{ctr} is a pair $(j, S(j))$.

Algorithm	Read	Write (Dense)	Write (Sparse)
Row-wise	$\sum n_i$	dN	$\sum n_i$
Column-wise	$\sum n_i$	d	
Column-to-row	$\sum n_i^2$		

Figure 6: Per Epoch Execution Cost of Row- and Column-wise Access. The Write column is for a single model replica. Given a dataset $A \in \mathbb{R}^{N \times d}$, let n_i be the number of non-zero elements a_i.

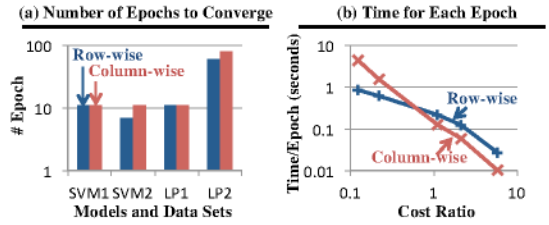

Figure 7: Illustration of the Method Selection Tradeoff. (a) These four datasets are RCV1, Reuters, Amazon, and Google, respectively. (b) The "cost ratio" is defined as the ratio of costs estimated for row-wise and column-wise methods: $(1+\alpha)\sum_i n_i/(\sum_i n_i^2+\alpha d)$, where n_i is the number of non-zero elements of i^{th} row of A and α is the cost ratio between writing and reads. We set $\alpha = 10$ to plot this graph.

Tradeoffs. We consider the two tradeoffs that we use for a simple cost model (Figure 6). Let n_i be the number of non-zeros in row i; when we store the data as sparse vectors/matrices in CSR format, the number of reads in a row-wise access method is $\sum_{i=1}^{N} n_i$. Since each example is likely to be written back in a dense write, we perform dN writes per epoch. Our cost model combines these two costs linearly with a factor α that accounts for writes being more expensive, on average, because of contention. The factor α is estimated at installation time by measuring on a small set of datasets. The parameter α is in 4 to 12 and grows with the number of sockets; e.g., for local2, $\alpha \approx 4$, and for local8, $\alpha \approx 12$. Thus, α may increase in the future.

Statistical Efficiency. We observe that each access method has comparable *statistical efficiency*. To illustrate this, we run all methods on all of our datasets and report the number of epochs that one method converges to a given error to the optimal loss, and Figure 7(a) shows the result on four datasets with 10% error. We see that the gap in the number of epochs across different methods is small (always within 50% of each other).

Hardware Efficiency. Different access methods can change the time per epoch by up to a factor of 10×, and there is a cross-over point. To see this, we run both methods on a series of synthetic datasets where we control the number of non-zero elements per row by subsampling each row on the Music dataset (see Section 4 for more details). For each subsampled dataset, we plot the cost ratio on the x-axis, and we plot their actual running time per epoch in Figure 7(b). We see a cross-over point on the time used per epoch: when the cost ratio is small, row-wise outperforms column-wise by 6×, as the column-wise method reads more data; on the other hand, when the ratio is large, the column-wise method outperforms the row-wise method by 3×, as the column-wise method has lower write contention. We observe similar cross-over points on our other datasets.

Cost-based Optimizer. DimmWitted estimates the execution time of different access methods using the number of bytes that each method reads and writes in one epoch, as shown in Figure 6. For writes, it is slightly more complex: for models such as SVM, each gradient step in row-wise access only updates the coordinates where the input vector contains non-zero elements. We call this scenario a *sparse* update; otherwise, it is a *dense* update.

DimmWitted needs to estimate the ratio of the cost of reads to writes. To do this, it runs a simple benchmark dataset. We find that, for all the eight datasets, five statistical models, and five machines that we use in the experiments, the cost model is robust to this parameter: as long as writes are 4× to 100× more expensive than reading, the cost model makes the correct decision between row-wise and column-wise access.

3.3 Model Replication

In DimmWitted, we consider three model replication strategies. The first two strategies, namely PerCore and PerMachine, are similar to traditional shared-nothing and shared-memory architecture, respectively. We also consider a hybrid strategy, PerNode, designed for NUMA machines.

3.3.1 Granularity of Model Replication

The difference between the three model replication strategies is the granularity of replicating a model. We first describe PerCore and PerMachine and their relationship with other existing systems (Figure 5). We then describe PerNode, a simple, novel hybrid strategy that we designed to leverage the structure of NUMA machines.

PerCore. In the PerCore strategy, each core maintains a mutable state, and these states are combined to form a new version of the model (typically at the end of each epoch). This is essentially a shared-nothing architecture; it is implemented in Impala, Pivotal, and Hadoop-based frameworks. PerCore is popularly implemented by state-of-the-art statistical analytics frameworks such as Bismarck, Spark, and GraphLab. There are subtle variations to this approach: in Bismarck's implementation, each worker processes a partition of the data, and its model is averaged at the end of each epoch; Spark implements a minibatch-based approach in which parallel workers calculate the gradient based on examples, and then gradients are aggregated by a single thread to update the final model; GraphLab implements an event-based approach where each different task is dynamically scheduled to satisfy the given consistency requirement. In DimmWitted, we implement PerCore in a way that is similar to Bismarck, where each worker has its own model replica, and each worker is responsible for updating its replica.[3] As we will show in the experiment section, DimmWitted's implementation is 3-100× faster than either

[3]We implemented MLlib's minibatch in DimmWitted. We find that the Hogwild!-like implementation always dominates the minibatch implementation. DimmWitted's column-wise implementation for PerMachine is similar to GraphLab, with the only difference that DimmWitted does not schedule the task in an event-driven way.

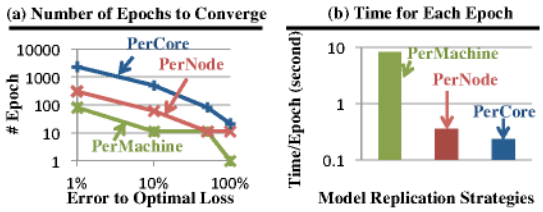

Figure 8: Illustration of Model Replication.

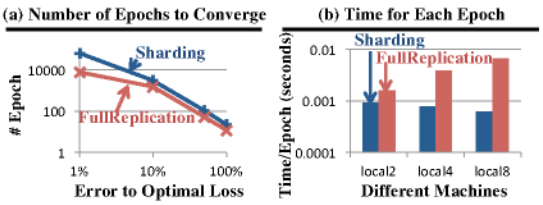

Figure 9: Illustration of Data Replication.

GraphLab and Spark. Both systems have additional sources of overhead that DimmWitted does not, e.g., for fault tolerance in Spark and a distributed environment in both. We are not making an argument about the relative merits of these features in applications, only that they would obscure the tradeoffs that we study in this paper.

PerMachine. In the PerMachine strategy, there is a single model replica that all workers update during execution. PerMachine is implemented in Hogwild! and Google's Downpour. Hogwild! implements a lock-free protocol, which forces the hardware to deal with coherence. Although different writers may overwrite each other and readers may have dirty reads, Niu et al. [25] prove that Hogwild! converges.

PerNode. The PerNode strategy is a hybrid of PerCore and PerMachine. In PerNode, each NUMA node has a single model replica that is shared among all cores on that node.

Model Synchronization. Deciding how often the replicas synchronize is key to the design. In Hadoop-based and Bismarck-based models, they synchronize at the end of each epoch. This is a shared-nothing approach that works well in user-defined aggregations. However, we consider finer granularities of sharing. In DimmWitted, we chose to have one thread that periodically reads models on all other cores, averages their results, and updates each replica.

One key question for model synchronization is *how frequently should the model be synchronized?* Intuitively, we might expect that more frequent synchronization will lower the throughput; on the other hand, the more frequently we synchronize, the fewer number of iterations we might need to converge. However, in DimmWitted, we find that the optimal choice is to communicate as frequently as possible. The intuition is that the QPI has staggering bandwidth (25GB/s) compared to the small amount of data we are shipping (megabytes). As a result, in DimmWitted, we implement an asynchronous version of the model averaging protocol: a separate thread averages models, with the effect of batching many writes together across the cores into one write, reducing the number of stalls.

Tradeoffs. We observe that PerNode is more hardware efficient, as it takes less time to execute an epoch than PerMachine; PerMachine might use fewer number of epochs to converge than PerNode.

Statistical Efficiency. We observe that PerMachine usually takes fewer epochs to converge to the same loss compared to PerNode, and PerNode uses fewer number of epochs than PerCore. To illustrate this observation, Figure 8(a) shows the number of epochs that each strategy requires to converge to a given loss for SVM (RCV1). We see that PerMachine always uses the least number of epochs to converge to a given loss: intuitively, the single model replica has more information at each step, which means that there is less redundant work. We observe similar phenomena when comparing PerCore and PerNode.

Hardware Efficiency. We observe that PerNode uses much less time to execute an epoch than PerMachine. To illustrate the difference in the time that each model replication strategy uses to finish one epoch, we show in Figure 8(b) the execution time of three strategies on SVM (RCV1). We see that PerNode is 23× faster than PerMachine and that PerCore is 1.5× faster than PerNode. PerNode takes advantage of the locality provided by the NUMA architecture. Using PMUs, we find that PerMachine incurs 11× more cross-node DRAM requests than PerNode.

Rule of Thumb. For SGD-based models, PerNode usually gives optimal results, while for SCD-based models, PerMachine does. Intuitively, this is caused by the fact that SGD has a denser update pattern than SCD, so, PerMachine suffers from hardware efficiency.

3.4 Data Replication

In DimmWitted, each worker processes a subset of data and then updates its model replica. To assign a subset of data to each worker, we consider two strategies.

Sharding. Sharding is a popular strategy implemented in systems such as Hogwild!, Spark, and Bismarck, in which the dataset is partitioned, and each worker only works on its partition of data. When there is a single model replica, Sharding avoids wasted computation, as each tuple is processed once per epoch. However, when there are multiple model replicas, Sharding might increase the variance of the estimate we form on each node, lowering the statistical efficiency. In DimmWitted, we implement Sharding by randomly partitioning the rows (resp. columns) of a data matrix for the row-wise (resp. column-wise) access method. In column-to-row access, we also replicate other rows that are needed.

FullReplication. A simple alternative to Sharding is FullReplication, in which we replicate the whole dataset many times (PerCore or PerNode). In PerNode, each NUMA node will have a full copy of the data. Each node accesses its data in a different order, which means that the replicas provide non-redundant statistical information. Statistically, there are two benefits of FullReplication: (1) averaging different estimates from each node has a lower variance, and (2) the estimate at each node has lower variance than in the Sharding case, as each node's estimate is based on the whole data. From a hardware efficiency perspective, reads are more fre-

Model	Dataset	#Row	#Col.	NNZ	Size (Sparse)	Size (Dense)	Sparse
SVM LR LS	RCV1	781K	47K	60M	914MB	275GB	✔
	Reuters	8K	18K	93K	1.4MB	1.2GB	✔
	Music	515K	91	46M	701MB	0.4GB	
	Forest	581K	54	30M	490MB	0.2GB	
LP	Amazon	926K	335K	2M	28MB	>1TB	✔
	Google	2M	2M	3M	25MB	>1TB	✔
QP	Amazon	1M	1M	7M	104MB	>1TB	✔
	Google	2M	2M	10M	152MB	>1TB	✔
Gibbs	Paleo	69M	30M	108M	2GB	>1TB	✔
NN	MNIST	120M	800K	120M	2GB	>1TB	✔

Figure 10: Dataset Statistics. NNZ refers to the Number of Non-zero elements. The # columns is equal to the number of variables in the model.

quent from local NUMA memory in PerNode than in PerMachine. The PerNode approach dominates the PerCore approach, as reads from the same node go to the same NUMA memory. Thus, we do not consider PerCore replication from this point on.

Tradeoffs. Not surprisingly, we observe that FullReplication takes more time for each epoch than Sharding. However, we also observe that FullReplication uses fewer epochs than Sharding, especially to achieve low error. We illustrate these two observations by showing the result of running SVM on Reuters using PerNode in Figure 9.

Statistical Efficiency. FullReplication uses fewer epochs, especially to low-error tolerance. Figure 9(a) shows the number of epochs that each strategy takes to converge to a given loss. We see that, for within 1% of the loss, FullReplication uses 10× fewer epochs on a two-node machine. This is because each model replica sees more data than Sharding, and therefore has a better estimate. Because of this difference in the number of epochs, FullReplication is 5× faster in wall-clock time than Sharding to converge to 1% loss. However, we also observe that, at high-error regions, FullReplication uses more epochs than Sharding and causes a comparable execution time to a given loss.

Hardware Efficiency. Figure 9(b) shows the time for each epoch across different machines with different numbers of nodes. Because we are using the PerNode strategy, which is the optimal choice for this dataset, the more nodes a machine has, the slower FullReplication is for each epoch. The slow-down is roughly consistent with the number of nodes on each machine. This is not surprising because each epoch of FullReplication processes more data than Sharding.

4. EXPERIMENTS

We validate that exploiting the tradeoff space that we described enables DimmWitted's orders of magnitude speedup over state-of-the-art competitor systems. We also validate that each tradeoff discussed in this paper affects the performance of DimmWitted.

4.1 Experiment Setup

We describe the details of our experimental setting.

Datasets and Statistical Models. We validate the performance and quality of DimmWitted on a diverse set of statistical models and datasets. For statistical models, we choose five models that are among the most popular models used in statistical analytics: (1) Support Vector Machine (SVM), (2) Logistic Regression (LR), (3) Least Squares Regression (LS), (4) Linear Programming (LP), and (5) Quadratic Programming (QP). For each model, we choose datasets with different characteristics, including size, sparsity, and under- or over-determination. For SVM, LR, and LS, we choose four datasets: Reuters[4], RCV1[5], Music[6], and Forest.[7] Reuters and RCV1 are datasets for text classification that are sparse and underdetermined. Music and Forest are standard benchmark datasets that are dense and overdetermined. For QP and LR, we consider a social-network application, i.e., network analysis, and use two datasets from Amazon's customer data and Google's Google+ social networks.[8] Figure 10 shows the dataset statistics.

Metrics. We measure the quality and performance of DimmWitted and other competitors. To measure the quality, we follow prior art and use the loss function for all functions. For end-to-end performance, we measure the wall-clock time it takes for each system to converge to a loss that is within 100%, 50%, 10%, and 1% of the optimal loss.[9] When measuring the wall-clock time, we do not count the time used for data loading and result outputting for all systems. We also use other measurements to understand the details of the tradeoff space, including (1) local LLC request, (2) remote LLC request, and (3) local DRAM request. We use Intel Performance Monitoring Units (PMUs) and follow the manual[10] to conduct these experiments.

Experiment Setting. We compare DimmWitted with four competitor systems: GraphLab [21], GraphChi [17], MLlib [33] over Spark [37], and Hogwild! [25]. GraphLab is a distributed graph processing system that supports a large range of statistical models. GraphChi is similar to GraphLab but with a focus on multi-core machines with secondary storage. MLlib is a package of machine learning algorithms implemented over Spark, an in-memory implementation of the MapReduce framework. Hogwild! is an in-memory lock-free framework for statistical analytics. We find that all four systems pick some points in the tradeoff space that we considered in DimmWitted. In GraphLab and GraphChi, all models are implemented using stochastic coordinate descent (column-wise access); in MLlib and Hogwild!, SVM and LR are implemented using stochastic gradient descent (row-wise access). We use implementations that are provided by the original developers whenever possible. For models without code provided by the developers, we only change the corresponding gradient function.[11] For GraphChi, if the

[4] archive.ics.uci.edu/ml/datasets/Reuters-21578+Text+Categorization+Collection
[5] about.reuters.com/researchandstandards/corpus/
[6] archive.ics.uci.edu/ml/datasets/YearPredictionMSD
[7] archive.ics.uci.edu/ml/datasets/Covertype
[8] snap.stanford.edu/data/
[9] We obtain the optimal loss by running all systems for one hour and choosing the lowest.
[10] software.intel.com/en-us/articles/performance-monitoring-unit-guidelines
[11] For sparse models, we change the dense vector data structure in MLlib to a sparse vector, which only improves its performance.

Dataset		Within 1% of the Optimal Loss					Within 50% of the Optimal Loss				
		GraphLab	GraphChi	MLlib	Hogwild!	DW	GraphLab	GraphChi	MLlib	Hogwild!	DW
SVM	Reuters	58.9	56.7	15.5	**0.1**	**0.1**	13.6	11.2	0.6	**0.01**	**0.01**
	RCV1	> 300.0	> 300.0	> 300	61.4	**26.8**	> 300.0	> 300.0	58.0	0.71	**0.17**
	Music	> 300.0	> 300.0	156	33.32	**23.7**	31.2	27.1	7.7	0.17	**0.14**
	Forest	16.2	15.8	2.70	0.23	**0.01**	1.9	1.4	0.15	0.03	**0.01**
LR	Reuters	36.3	34.2	19.2	**0.1**	**0.1**	13.2	12.5	1.2	**0.03**	**0.03**
	RCV1	> 300.0	> 300.0	> 300.0	38.7	**19.8**	> 300.0	> 300.0	68.0	0.82	**0.20**
	Music	> 300.0	> 300.0	> 300.0	35.7	**28.6**	30.2	28.9	8.9	0.56	**0.34**
	Forest	29.2	28.7	3.74	0.29	**0.03**	2.3	2.5	0.17	0.02	**0.01**
LS	Reuters	132.9	121.2	92.5	4.1	**3.2**	16.3	16.7	1.9	0.17	**0.09**
	RCV1	> 300.0	> 300.0	> 300	27.5	**10.5**	> 300.0	> 300.0	32.0	1.30	**0.40**
	Music	> 300.0	> 300.0	221	40.1	**25.8**	> 300.0	> 300.0	11.2	0.78	**0.52**
	Forest	25.5	26.5	1.01	0.33	**0.02**	2.7	2.9	0.15	0.04	**0.01**
LP	Amazon	2.7	2.4	> 120.0	> 120.0	**0.94**	2.7	2.1	120.0	1.86	**0.94**
	Google	13.4	11.9	> 120.0	> 120.0	**12.56**	2.3	2.0	120.0	3.04	**2.02**
QP	Amazon	6.8	5.7	> 120.0	> 120.0	**1.8**	6.8	5.7	> 120.0	> 120.00	**1.50**
	Google	12.4	10.1	> 120.0	> 120.0	**4.3**	9.9	8.3	> 120.0	> 120.00	**3.70**

Figure 11: End-to-End Comparison (time in seconds). The column DW refers to DimmWitted. We take 5 runs on local2 and report the average (standard deviation for all numbers < 5% of the mean). Entries with > indicate a timeout.

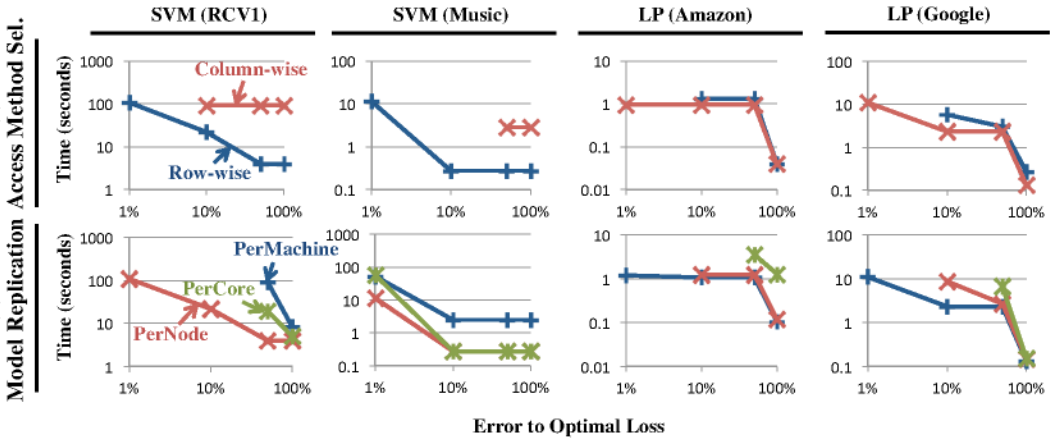

Figure 12: Tradeoffs in DimmWitted. Missing points timeout in 120 seconds.

corresponding model is implemented in GraphLab but not GraphChi, we follow GraphLab's implementation.

We run experiments on a variety of architectures. These machines differ in a range of configurations, including the number of NUMA nodes, the size of last-level cache (LLC), and memory bandwidth. See Figure 3 for a summary of these machines. DimmWitted, Hogwild!, GraphLab, and GraphChi are implemented using C++, and MLlib/Spark is implemented using Scala. We tune both GraphLab and MLlib according to their best practice guidelines.[12] For both GraphLab, GraphChi, and MLlib, we try different ways of increasing locality on NUMA machines, including trying to use `numactl` and implementing our own RDD for MLlib; there is more detail in the full version of this paper. Systems are compiled with g++ 4.7.2 (-O3), Java 1.7, or Scala 2.9.

4.2 End-to-End Comparison

We validate that DimmWitted outperforms competitor systems in terms of end-to-end performance and quality.

[12] MLlib: `spark.incubator.apache.org/docs/0.6.0/tuning.html`; GraphLab: `graphlab.org/tutorials-2/fine-tuning-graphlab-performance/`. For GraphChi, we tune the memory buffer size to ensure all data fit in memory and that there are no disk I/Os. We describe more detailed tuning for MLlib in the full version of this paper.

Note that both MLlib and GraphLab have extra overhead for fault tolerance, distributing work, and task scheduling. Our comparison between DimmWitted and these competitors is intended only to demonstrate that existing work for statistical analytics has not obviated the tradeoffs that we study here.

Protocol. For each system, we grid search their statistical parameters, including step size ({100.0,10.0,...,0.0001}) and mini-batch size for MLlib ({1%, 10%, 50%, 100%}); we always report the best configuration, which is essentially the same for each system. We measure the time it takes for each system to find a solution that is within 1%, 10%, and 50% of the optimal loss. Figure 11 shows the results for 1% and 50%; the results for 10% are similar. We report end-to-end numbers from local2, which has two nodes and 24 logical cores, as GraphLab does not run on machines with more than 64 logical cores. Figure 14 shows the DimmWitted's choice of point in the tradeoff space on local2.

As shown in Figure 11, DimmWitted always converges to the given loss in less time than the other competitors. On SVM and LR, DimmWitted could be up to 10× faster than Hogwild!, and more than two orders of magnitude faster than GraphLab and Spark. The difference between DimmWitted and Hogwild! is greater for LP and QP, where

DimmWitted outperforms Hogwild! by more than two orders of magnitude. On LP and QP, DimmWitted is also up to 3× faster than GraphLab and GraphChi, and two orders of magnitude faster than MLlib.

Tradeoff Choices. We dive more deeply into these numbers to substantiate our claim that there are some points in the tradeoff space that are not used by GraphLab, GraphChi, Hogwild!, and MLlib. Each tradeoff selected by our system is shown in Figure 14. For example, GraphLab and GraphChi uses column-wise access for all models, while MLlib and Hogwild! use row-wise access for all models and allow only PerMachine model replication. These special points work well for some but not all models. For example, for LP and QP, GraphLab and GraphChi are only 3× slower than DimmWitted, which chooses column-wise and PerMachine. This factor of 3 is to be expected, as GraphLab also allows distributed access and so has additional overhead. However there are other points: for SVM and LR, DimmWitted outperforms GraphLab and GraphChi, because the column-wise algorithm implemented by GraphLab and GraphChi is not as efficient as row-wise on the same dataset. DimmWitted outperforms Hogwild! because DimmWitted takes advantage of model replication, while Hogwild! incurs 11× more cross-node DRAM requests than DimmWitted; in contrast, DimmWitted incurs 11× more local DRAM requests than Hogwild! does.

For SVM, LR, and LS, we find that DimmWitted outperforms MLlib, primarily due to a different point in the tradeoff space. In particular, MLlib uses batch-gradient-descent with a PerCore implementation, while DimmWitted uses stochastic gradient and PerNode. We find that, for the Forest dataset, DimmWitted takes 60× fewer number of epochs to converge to 1% loss than MLlib. For each epoch, DimmWitted is 4× faster. These two factors contribute to the 240× speed-up of DimmWitted over MLlib on the Forest dataset (1% loss). MLlib has overhead for scheduling, so we break down the time that MLlibuses for scheduling and computation. We find that, for Forest, out of the total 2.7 seconds of execution, MLlib uses 1.8 seconds for computation and 0.9 seconds for scheduling. We also implemented a batch-gradient-descent and PerCore implementation inside DimmWitted to remove these and C++ versus Scala differences. The 60× difference in the number of epochs until convergence still holds, and our implementation is only 3× faster than MLlib. This implies that the main difference between DimmWitted and MLlib is the point in the tradeoff space—not low-level implementation differences.

For LP and QP, DimmWitted outperforms MLlib and Hogwild! because the row-wise access method implemented by these systems is not as efficient as column-wise access on the same data set. GraphLab does have column-wise access, so DimmWitted outperforms GraphLab and GraphChi because DimmWitted finishes each epoch up to 3× faster, primarily due to low-level issues. This supports our claims that the tradeoff space is interesting for analytic engines and that no one system has implemented all of them.

Throughput. We compare the throughput of different systems for an extremely simple task: parallel sums. Our implementation of parallel sum follows our implementation of other statistical models (with a trivial update function), and uses all cores on a single machine. Figure 13 shows the throughput on all systems on different models on one dataset. We see from Figure 13 that DimmWitted achieves the highest throughput of all the systems. For parallel sum, DimmWitted is 1.6× faster than Hogwild!, and we find that DimmWitted incurs 8× fewer LLC cache misses than Hogwild!. Compared with Hogwild!, in which all threads write to a single copy of the sum result, DimmWitted maintains one single copy of the sum result per NUMA node, so the workers on one NUMA node do not invalidate the cache on another NUMA node. When running on only a single thread, DimmWitted has the same implementation as Hogwild!. Compared with GraphLab and GraphChi, DimmWitted is 20× faster, likely due to the overhead of GraphLab and GraphChi dynamically scheduling tasks and/or maintaining the graph structure. To compare DimmWitted with MLlib, which is written in Scala, we implemented a Scala version, which is 3× slower than C++; this suggests that the overhead is not just due to the language. If we do not count the time that MLlibuses for scheduling and only count the time of computation, we find that DimmWitted is 15× faster than MLlib.

	SVM (RCV1)	LR (RCV1)	LS (RCV1)	LP (Google)	QP (Google)	Parallel Sum
GraphLab	0.2	0.2	0.2	0.2	0.1	0.9
GraphChi	0.3	0.3	0.2	0.2	0.2	1.0
MLlib	0.2	0.2	0.2	0.1	0.02	0.3
Hogwild!	1.3	1.4	1.3	0.3	0.2	13
DimmWitted	5.1	5.2	5.2	0.7	1.3	21

Figure 13: Comparison of Throughput (GB/seconds) of Different Systems on local2.

		Access Methods	Model Replication	Data Replication
SVM LR LS	Reuters RCV1 Music	Row-wise	PerNode	FullReplication
LP QP	Amazon Google	Column-wise	PerMachine	FullReplication

Figure 14: Plans that DimmWitted Chooses in the Tradeoff Space for Each Dataset on Machine local2.

4.3 Tradeoffs of DimmWitted

We validate that all the tradeoffs described in this paper have an impact on the efficiency of DimmWitted. We report on a more modern architecture, local4 with four NUMA sockets, in this section. We describe how the results change with different architectures.

4.3.1 Access Method Selection

We validate that different access methods have different performance, and that no single access method dominates the others. We run DimmWitted on all statistical models and compare two strategies, row-wise and column-wise. In each experiment, we force DimmWitted to use the corresponding access method, but report the best point for the other tradeoffs. Figure 12(a) shows the results as we measure the time it takes to achieve each loss. The more stringent loss requirements (1%) are on the left-hand side. The horizontal line segments in the graph indicate that a model may reach, say, 50% as quickly (in epochs) as it reaches 100%.

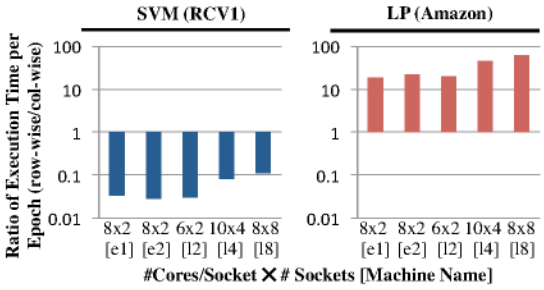

Figure 15: Ratio of Execution Time per Epoch (row-wise/column-wise) on Different Architectures. A number larger than 1 means that row-wise is slower. l2 means local2, e1 means ec2.1, etc.

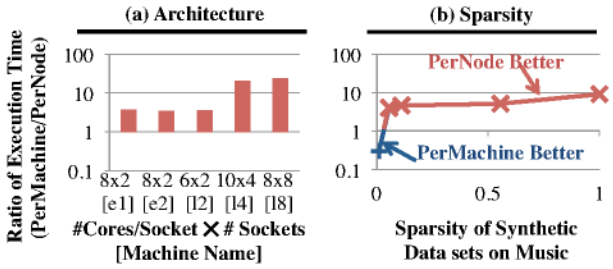

Figure 16: The Impact of Different Architectures and Sparsity on Model Replication. A ratio larger than 1 means that PerNode converges faster than PerMachine to 50% loss.

We see from Figure 12(a) that the difference between row-wise and column-to-row access could be more than 100× for different models. For SVM on RCV1, row-wise access converges at least 4× faster to 10% loss and at least 10× faster to 100% loss. We observe similar phenomena for Music; compared with RCV1, column-to-row access converges to 50% loss and 100% loss at a 10× slower rate. With such datasets, the column-to-row access simply requires more reads and writes. This supports the folk wisdom that gradient methods are preferable to coordinate descent methods. On the other hand, for LP, column-wise access dominates: row-wise access does not converge to 1% loss within the timeout period for either Amazon or Google. Column-wise access converges at least 10-100× faster than row-wise access to 1% loss. We observe that LR is similar to SVM and QP is similar to LP. Thus, no access method dominates all the others.

The cost of writing and reading are different and is captured by a parameter that we called α in Section 3.2. We describe the impact of this factor on the relative performance of row- and column-wise strategies. Figure 15 shows the ratio of the time that each strategy uses (row-wise/column-wise) for SVM (RCV1) and LP (Amazon). We see that, as the number of sockets on a machine increases, the ratio of execution time increases, which means that row-wise becomes slower relative to column-wise, i.e., with increasing α. As the write cost captures the cost of a hardware-resolved conflict, we see that this constant is likely to grow. Thus, if next-generation architectures increase in the number of sockets, the cost parameter α and consequently the importance of this tradeoff are likely to grow.

Cost-based Optimizer. We observed that, for all datasets, our cost-based optimizer selects row-wise access for SVM, LR, and LS, and column-wise access for LP and QP. These choices are consistent with what we observed in Figure 12.

4.3.2 Model Replication

We validate that there is no single strategy for model replication that dominates the others. We force DimmWitted to run strategies in PerMachine, PerNode, and PerCore and choose other tradeoffs by choosing the plan that achieves the best result. Figure 12(b) shows the results.

We see from Figure 12(b) that the gap between PerMachine and PerNode could be up to 100×. We first observe that PerNode dominates PerCore on all datasets. For SVM on RCV1, PerNode converges 10× faster than PerCore to 50% loss, and for other models and datasets, we observe a similar phenomenon. This is due to the low statistical efficiency of PerCore, as we discussed in Section 3.3. Although PerCore eliminates write contention inside one NUMA node, this write contention is less critical. For large models and machines with small caches, we also observe that PerCore could spill the cache.

These graphs show that neither PerMachine nor PerNode dominates the other across all datasets and statistical models. For SVM on RCV1, PerNode converges 12× faster than PerMachine to 50% loss. However, for LP on Amazon, PerMachine is at least 14× faster than PerNode to converge to 1% loss. For SVM, PerNode converges faster because it has 5× higher throughput than PerMachine, and for LP, PerNode is slower because PerMachine takes at least 10× fewer epochs to converge to a small loss. One interesting observation is that, for LP on Amazon, PerMachine and PerNode do have comparable performance to converge to 10% loss. Compared with the 1% loss case, this implies that PerNode's statistical efficiency decreases as the algorithm tries to achieve a smaller loss. This is not surprising, as one must reconcile the PerNode estimates.

We observe that the relative performance of PerMachine and PerNode depends on (1) the number of sockets used on each machine and (2) the sparsity of the update.

To validate (1), we measure the time that PerNode and PerMachine take on SVM (RCV1) to converge to 50% loss on various architectures, and we report the ratio (PerMachine/PerNode) in Figure 16. We see that PerNode's relative performance improves with the number of sockets. We attribute this to the increased cost of write contention in PerMachine.

To validate (2), we generate a series of synthetic datasets, each of which subsamples the elements in each row of the Music dataset; Figure 16(b) shows the results. When the sparsity is 1%, PerMachine outperforms PerNode, as each update touches only one element of the model; thus, the write contention in PerMachine is not a bottleneck. As the sparsity increases (i.e., the update becomes denser), we observe that PerNode outperforms PerMachine.

4.3.3 Data Replication

We validate the impact of different data replication strategies. We run DimmWitted by fixing data replication strategies to FullReplication or Sharding and choosing the best plan for each other tradeoff. We measure the execution time for

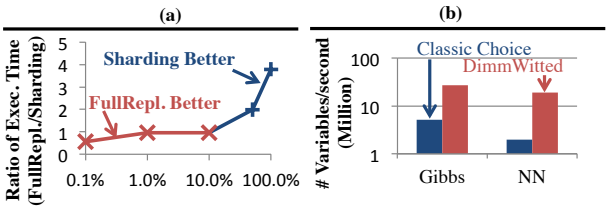

Figure 17: (a) Tradeoffs of Data Replication. A ratio smaller than 1 means that FullReplication is faster. (b) Performance of Gibbs Sampling and Neural Networks Implemented in DimmWitted.

each strategy to converge to a given loss for SVM on the same dataset, RCV1. We report the ratio of these two strategies as FullReplication/Sharding in Figure 17(a). We see that, for the low-error region (e.g., 0.1%), FullReplication is 1.8-2.5× faster than Sharding. This is because FullReplication decreases the skew of data assignment to each worker, so hence each individual model replica can form a more accurate estimate. For the high-error region (e.g., 100%), we observe that FullReplication appears to be 2-5× slower than Sharding. We find that, for 100% loss, both FullReplication and Sharding converge in a single epoch, and Sharding may therefore be preferred, as it examines less data to complete that single epoch. In all of our experiments, FullReplication is never substantially worse and can be dramatically better. Thus, if there is available memory, the FullReplication data replication seems to be preferable.

5. EXTENSIONS

We briefly describe how to run Gibbs sampling (which uses a column-to-row access method) and deep neural networks (which uses a row access method). Using the same tradeoffs, we achieve a significant increase in speed over the classical implementation choices of these algorithms. A more detailed description is in the full version of this paper.

5.1 Gibbs Sampling

Gibbs sampling is one of the most popular algorithms to solve statistical inference and learning over probabilistic graphical models [30]. We briefly describe Gibbs sampling over factor graphs and observe that its main step is a column-to-row access. A factor graph can be thought of as a bipartite graph of a set of variables and a set of factors. To run Gibbs sampling, the main operation is to select a single variable, and calculate the conditional probability of this variable, which requires the fetching of all factors that contain this variable and all assignments of variables connected to these factors. This operation corresponds to the column-to-row access method. Similar to first-order methods, recently, a Hogwild! algorithm for Gibbs was established [15]. As shown in Figure 17(b), applying the technique in DimmWitted to Gibbs sampling achieves 4× the throughput of samples as the PerMachine strategy.

5.2 Deep Neural Networks

Neural networks are one of the most classic machine learning models [22]; recently, these models have been intensively revisited by adding more layers [10, 18]. A deep neural network contains multiple layers, and each layer contains a set of neurons (variables). Different neurons connect with each other only by links across consecutive layers. The value of one neuron is a function of all the other neurons in the previous layer and a set of weights. Variables in the last layer have human labels as training data; the goal of deep neural network learning is to find the set of weights that maximizes the likelihood of the human labels. Back-propagation with stochastic gradient descent is the de facto method of optimizing a deep neural network.

Following LeCun et al. [19], we implement SGD over a seven-layer neural network with 0.12 billion neurons and 0.8 million parameters using a standard handwriting-recognition benchmark dataset called MNIST[13]. Figure 17(b) shows the number of variables that are processed by DimmWitted per second. For this application, DimmWitted uses PerNode and FullReplication, and the classical choice made by LeCun is PerMachine and Sharding. As shown in Figure 17(b), DimmWitted achieves more than an order of magnitude higher throughput than this classical baseline (to achieve the same quality as reported in this classical paper).

6. RELATED WORK

We review work in four main areas: statistical analytics, data mining algorithms, shared-memory multiprocessors optimization, and main-memory databases. We include more extensive related work in the full version of this paper.

Statistical Analytics. There is a trend to integrate statistical analytics into data processing systems. Database vendors have recently put out new products in this space, including Oracle, Pivotal's MADlib [13], IBM's SystemML [12], and SAP's HANA. These systems support statistical analytics in existing data management systems. A key challenge for statistical analytics is performance.

A handful of data processing frameworks have been developed in the last few years to support statistical analytics, including Mahout for Hadoop, MLI for Spark [33], GraphLab [21], and MADLib for PostgreSQL or Greenplum [13]. Although these systems increase the performance of corresponding statistical analytics tasks significantly, we observe that each of them implements one point in DimmWitted's tradeoff space. DimmWitted is not a system; our goal is to study this tradeoff space.

Data Mining Algorithms. There is a large body of data mining literature regarding how to optimize various algorithms to be more architecturally aware [26, 38, 39]. Zaki et al. [26,39] study the performance of a range of different algorithms, including associated rule mining and decision tree on shared-memory machines, by improving memory locality and data placement in the granularity of cachelines, and decreasing the cost of coherent maintenance between multiple CPU caches. Ghoting et al. [11] optimize the cache behavior of frequent pattern mining using novel cache-conscious techniques, including spatial and temporal locality, prefetching, and tiling. Jin et al. [14] discuss tradeoffs in replication and locking schemes for K-means, association rule mining, and neural nets. This work considers the hardware efficiency of the algorithm, but not statistical efficiency, which is the focus of DimmWitted. In addition, Jin et al. do not consider lock-free execution, a key aspect of this paper.

Shared-memory Multiprocessor Optimization. Performance optimization on shared-memory multiprocessors machines is a classical topic. Anderson and Lam [3] and

[13] `yann.lecun.com/exdb/mnist/`

Carr et al.'s [7] seminal work used complier techniques to improve locality on shared-memory multiprocessor machines. DimmWitted's *locality group* is inspired by Anderson and Lam's discussion of *computation decomposition* and *data decomposition*. These locality groups are the centerpiece of the Legion project [5]. In recent years, there have been a variety of *domain specific languages* (DSLs) to help the user extract parallelism; two examples of these DSLs include Galois [23, 24] and OptiML [35] for Delite [8]. Our goals are orthogonal: these DSLs require knowledge about the tradeoffs of the hardware, such as those provided by our study.

Main-memory Databases. The database community has recognized that multi-socket, large-memory machines have changed the data processing landscape, and there has been a flurry of recent work about how to build in-memory analytics systems [2,4,9,16,20,27,28,36]. Classical tradeoffs have been revisited on modern architectures to gain significant improvement: Balkesen et al. [4], Albutiu et al. [2], Kim et al. [16], and Li [20] study the tradeoff for joins and shuffling, respectively. This work takes advantage of modern architectures, e.g., NUMA and SIMD, to increase memory bandwidth. We study a new tradeoff space for statistical analytics in which the performance of the system is affected by both hardware efficiency and statistical efficiency.

7. CONCLUSION

For statistical analytics on main-memory, NUMA-aware machines, we studied tradeoffs in access methods, model replication, and data replication. We found that using novel points in this tradeoff space can have a substantial benefit: our DimmWitted prototype engine can run at least one popular task at least 100× faster than other competitor systems. This comparison demonstrates that this tradeoff space may be interesting for current and next-generation statistical analytics systems.

Acknowledgments We would like to thank Arun Kumar, Victor Bittorf, the Delite team, the Advanced Analytics at Oracle, Greenplum/Pivotal, and Impala's Cloudera team for sharing their experiences in building analytics systems. We gratefully acknowledge the support of the Defense Advanced Research Projects Agency (DARPA) XDATA Program under No. FA8750-12-2-0335 and the DEFT Program under No. FA8750-13-2-0039, the National Science Foundation (NSF) CAREER Award under No. IIS-1353606, the Office of Naval Research (ONR) under awards No. N000141210041 and No. N000141310129, the Sloan Research Fellowship, American Family Insurance, Google, and Toshiba. Any opinions, findings, and conclusion or recommendations expressed in this material are those of the authors and do not necessarily reflect the view of DARPA, NSF, ONR, or the US government.

8. REFERENCES

[1] A. Agarwal, O. Chapelle, M. Dudík, and J. Langford. A reliable effective terascale linear learning system. *ArXiv e-prints*, 2011.

[2] M.-C. Albutiu, A. Kemper, and T. Neumann. Massively parallel sort-merge joins in main memory multi-core database systems. *PVLDB*, pages 1064–1075, 2012.

[3] J. M. Anderson and M. S. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *PLDI*, pages 112–125, 1993.

[4] C. Balkesen and et al. Multi-core, main-memory joins: Sort vs. hash revisited. *PVLDB*, pages 85–96, 2013.

[5] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken. Legion: expressing locality and independence with logical regions. In *SC*, page 66, 2012.

[6] L. Bergstrom. Measuring NUMA effects with the STREAM benchmark. *ArXiv e-prints*, 2011.

[7] S. Carr, K. S. McKinley, and C.-W. Tseng. Compiler optimizations for improving data locality. In *ASPLOS*, 1994.

[8] H. Chafi, A. K. Sujeeth, K. J. Brown, H. Lee, A. R. Atreya, and K. Olukotun. A domain-specific approach to heterogeneous parallelism. In *PPOPP*, pages 35–46, 2011.

[9] C. Chasseur and J. M. Patel. Design and evaluation of storage organizations for read-optimized main memory databases. *PVLDB*, pages 1474–1485, 2013.

[10] J. Dean and et al. Large scale distributed deep networks. In *NIPS*, pages 1232–1240, 2012.

[11] A. Ghoting and et al. Cache-conscious frequent pattern mining on modern and emerging processors. *VLDBJ*, 2007.

[12] A. Ghoting and et al. SystemML: Declarative machine learning on MapReduce. In *ICDE*, pages 231–242, 2011.

[13] J. M. Hellerstein and et al. The MADlib analytics library: Or MAD skills, the SQL. *PVLDB*, pages 1700–1711, 2012.

[14] R. Jin, G. Yang, and G. Agrawal. Shared memory parallelization of data mining algorithms: Techniques, programming interface, and performance. *TKDE*, 2005.

[15] M. J. Johnson, J. Saunderson, and A. S. Willsky. Analyzing Hogwild parallel Gaussian Gibbs sampling. In *NIPS*, 2013.

[16] C. Kim and et al. Sort vs. hash revisited: Fast join implementation on modern multi-core CPUs. *PVLDB*, 2009.

[17] A. Kyrola, G. Blelloch, and C. Guestrin. Graphchi: Large-scale graph computation on just a pc. In *OSDI*, pages 31–46, 2012.

[18] Q. V. Le and et al. Building high-level features using large scale unsupervised learning. In *ICML*, pages 8595–8598, 2012.

[19] Y. LeCun and et al. Gradient-based learning applied to document recognition. *IEEE*, pages 2278–2324, 1998.

[20] Y. Li and et al. NUMA-aware algorithms: the case of data shuffling. In *CIDR*, 2013.

[21] Y. Low and et al. Distributed GraphLab: A framework for machine learning in the cloud. *PVLDB*, pages 716–727, 2012.

[22] T. M. Mitchell. *Machine Learning*. McGraw-Hill, USA, 1997.

[23] D. Nguyen, A. Lenharth, and K. Pingali. A lightweight infrastructure for graph analytics. In *SOSP*, 2013.

[24] D. Nguyen, A. Lenharth, and K. Pingali. Deterministic Galois: On-demand, portable and parameterless. In *ASPLOS*, 2014.

[25] F. Niu and et al. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *NIPS*, pages 693–701, 2011.

[26] S. Parthasarathy, M. J. Zaki, M. Ogihara, and W. Li. Parallel data mining for association rules on shared memory systems. *Knowl. Inf. Syst.*, pages 1–29, 2001.

[27] L. Qiao and et al. Main-memory scan sharing for multi-core CPUs. *PVLDB*, pages 610–621, 2008.

[28] V. Raman and et al. DB2 with BLU acceleration: So much more than just a column store. *PVLDB*, pages 1080–1091, 2013.

[29] P. Richtárik and M. Takáč. Parallel coordinate descent methods for big data optimization. *ArXiv e-prints*, 2012.

[30] C. P. Robert and G. Casella. *Monte Carlo Statistical Methods (Springer Texts in Statistics)*. Springer, USA, 2005.

[31] A. Smola and S. Narayanamurthy. An architecture for parallel topic models. *PVLDB*, pages 703–710, 2010.

[32] S. Sonnenburg and et al. The SHOGUN machine learning toolbox. *J. Mach. Learn. Res.*, pages 1799–1802, 2010.

[33] E. Sparks and et al. MLI: An API for distributed machine learning. In *ICDM*, pages 1187–1192, 2013.

[34] S. Sridhar and et al. An approximate, efficient LP solver for LP rounding. In *NIPS*, pages 2895–2903, 2013.

[35] A. K. Sujeeth and et al. OptiML: An Implicitly Parallel Domain-Specific Language for Machine Learning. In *ICML*, pages 609–616, 2011.

[36] S. Tu and et al. Speedy transactions in multicore in-memory databases. In *SOSP*, pages 18–32, 2013.

[37] M. Zaharia and et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.

[38] M. Zaki and et al. Parallel classification for data mining on shared-memory multiprocessors. In *ICDE*, pages 198–205, 1999.

[39] M. J. Zaki and et al. New algorithms for fast discovery of association rules. In *KDD*, pages 283–286, 1997.