Scalable Asynchronous Gradient Descent Optimization for Out-of-Core Models

Chengjie Qin[‡*] Martin Torres[†] Florin Rusu[†]
[†]University of California Merced, [‡]GraphSQL, Inc.
{cqin3, mtorres58, frusu}@ucmerced.edu

ABSTRACT

Existing data analytics systems have approached predictive model training exclusively from a data-parallel perspective. Data examples are partitioned to multiple workers and training is executed concurrently over different partitions, under various synchronization policies that emphasize speedup or convergence. Since models with millions and even billions of features become increasingly common nowadays, model management becomes an equally important task for effective training. In this paper, we present a general framework for parallelizing stochastic optimization algorithms over massive models that cannot fit in memory. We extend the lock-free HOGWILD!-family of algorithms to disk-resident models by vertically partitioning the model offline and asynchronously updating the resulting partitions online. Unlike HOGWILD!, concurrent requests to the common model are minimized by a preemptive push-based sharing mechanism that reduces the number of disk accesses. Experimental results on real and synthetic datasets show that the proposed framework achieves improved convergence over HOGWILD! and is the only solution scalable to massive models.

1. INTRODUCTION

Data analytics is a major topic in contemporary data management and machine learning. Many platforms, e.g., OptiML [38], GraphLab [26], SystemML [12], Vowpal Wabbit [1], SimSQL [2], GLADE [4], Tupleware [7] and libraries, e.g., MADlib [15], Bismarck [11], MLlib [37], Mahout[1], have been proposed to provide support for parallel statistical analytics. Stochastic gradient descent is the most popular optimization method used to train analytics models across all these systems. It is implemented – in a form or another – by all of them. The seminal HOGWILD!-family of algorithms [28] for stochastic gradient descent has received – in particular – significant attention due to its near-linear speedups across a variety of machine learning tasks, but, mostly, because of its simplicity. Several studies have applied HOGWILD! to parallelize classical learning methods [33, 11, 25, 9, 8, 5] by performing model updates concurrently and asynchronously without locks.

Due to the explosive growth in data acquisition, the current trend is to devise prediction models with an ever-increasing number of features, i.e., big models. For example, Google has reported models with billions of features for predicting ad click-through rates [27] as early as 2013. Feature vectors with 25 billion unigrams and 218 billion bigrams are constructed for text analysis of the English Wikipedia corpus in [20]. Big models also appear in recommender systems. Spotify applies Low-rank Matrix Factorization (LMF) for 24 million users and 20 million songs[2], which leads to 4.4 billion features at a relatively small rank of 100.

Since HOGWILD! is an in-memory algorithm, it cannot handle these big models – models that go beyond the available memory of the system – directly. In truth, none of the analytics systems mentioned above support out-of-memory models because they represent the model as a single shared variable—not as a partitioned dataset, which is the case for the training data. The only exception is the sequential dot-product join operator introduced in [32] which represents the model as a relational table. Parameter Server [22] is an indirect approach that resorts to distributed shared memory. The big model is partitioned across several servers, with each server storing a sufficiently small model partition that fits in its local memory. In addition to the complexity incurred by model partitioning and replication across servers, Parameter Server also has a high cost in hardware and network traffic. While one can argue that memory will never be a problem in the cloud, this is not the case in IoT settings. The edge and fog computing paradigms[3] push processing to the devices acquiring the data which have rather scarce resources and do not consider data transfer a viable alternative—for bandwidth and privacy reasons. Machine learning training in such an environment has to consider secondary storage, e.g., disk, SSD, and flash cards, for storing big models.

Problem. In this work, we investigate parallel stochastic optimization methods for big models that cannot fit in memory. Specifically, we focus on designing a scalable HOGWILD! algorithm. Our setting is a single multi-core server with attached storage, i.e., disk(s). There is a worker thread associated with each core in the system. The training data as well as the model are stored on disk and moved into memory only when accessed. Training data are partitioned into chunks that are accessed and processed as a unit. Several chunks are processed concurrently by multiple worker threads—data-parallel processing. While access to the training data follows a well-behaved sequential pattern, the access to the model is unpredictable. Moreover, there are many model accesses for each training example—the number of non-zero entries in the example.

[*]Work mostly done while a Ph.D. student at UC Merced.
[1]https://mahout.apache.org

[2]http://slideshare.net/MrChrisJohnson/algorithmic-music-recommendations-at-spotify
[3]https://techcrunch.com/2016/08/02/how-fog-computing-pushes-iot-intelligence-to-the-edge/

Proceedings of the VLDB Endowment, Vol. 10, No. 10

Thus, the challenge in handling big models is how to efficiently schedule access to the model. In the worst case, each model access requires a disk access. This condition is worsened in data-parallel processing by the fact that multiple model accesses are made concurrently by the worker threads—model-parallel processing.

Approach. While extending HOGWILD! to disk-resident models is straightforward, designing a truly scalable algorithm that supports model and data-parallel processing is a considerably more challenging task. At a high-level, our approach targets the main source that impacts performance – the massive number of concurrent model accesses – with two classical database processing techniques—*vertical partitioning* and *model access sharing*.

The model is vertically partitioned offline based on the concept of "feature occurrence" – we say a feature "occurs" when it has a non-zero value in a training example – such that features that co-occur together require a single model access. Feature co-occurrence is a common characteristic of big models in many analytics tasks. For example, textual features such as n-grams[4] which extract a contiguous sequence of n words from text generate co-occurring features for commonly used sentences. Gene sequence patterns represent an even more widespread example in this category. It is important to notice that feature co-occurrence is fundamentally different from the feature correlation that standard feature engineering processes [24, 41, 6, 17] try to eliminate. In feature engineering, correlation between features is measured by coefficients such as Pearson's coefficient [13] instead of co-occurrence. In this work, we are interested exclusively in what features co-appear together. Thus, we refer to "feature co-occurrence" as "correlation".

During online training, access sharing is maximized at several stages in the processing hierarchy in order to reduce the number of disk-level model accesses. The data examples inside a chunk are logically partitioned vertically according to the model partitions generated offline. The goal of this stage is to cluster together accesses to model features even across examples—vertical partitioning achieves this only for the features that co-occur in the same example. In order to guarantee that access sharing occurs across partitions, we introduce a novel push-based mechanism to enforce sharing by vertical traversals of the example data and partial dot-product materialization. Workers preemptively push the features they acquire to all the other threads asynchronously. This is done only for read accesses. The number of write accesses is minimized by executing model updates at batch-level, rather than for every example. This technique, i.e., HogBatch [31, 35], is shown to dramatically increase the speedup of HOGWILD! – and its convergence – for memory-resident models because it eliminates the "pingpong" effect [35] on cache-coherent architectures.

Contributions. We design a scalable model and data-parallel framework for parallelizing stochastic optimization algorithms over big models. The framework organizes processing in two separate stages – offline model partitioning and asynchronous online training – and brings the following major contributions:

- Formalize model partitioning as vertical partitioning and design a scalable frequency-based model vertical partitioning algorithm. The resulting partitions are mapped to a novel composite key-value storage scheme.
- Devise an asynchronous method to traverse vertically the training examples in all the data partitions according to the model partitions generated offline.
- Design a push-based model sharing mechanism for incremental gradient computation based on partial dot-products.
- Implement the entire framework using User-Defined Aggregates (UDA) which provides generality across databases.
- Evaluate the framework for three analytics tasks over synthetic and real datasets. The results prove the scalability, reduced overhead incurred by model partitioning, and the consistent superior performance of the framework over an optimized HOGWILD! extension to big models.

Outline. Preliminaries on stochastic optimization and vertical partitioning are presented in Section 2, while HOGWILD! is introduced in Section 3. The high-level approach of the proposed framework is presented in Section 4, while the details are given in Section 5 (offline stage) and Section 6 (online stage). Experimental results are included in Section 7, related work in Section 8, while concluding remarks and plans for future work are in Section 9.

2. PRELIMINARIES

In this section, we give an overview of several topics relevant to the management and processing of big models. Specifically, we discuss gradient descent optimization as the state-of-the-art in big model training, key-value stores as the standard big model storage manager, and vertical partitioning.

Big model training. Consider the following optimization problem with a linearly separable objective function:

$$\Lambda(\vec{w}) = min_{w \in \mathbb{R}^d} \sum_{i=1}^{N} f\left(\vec{w}, \vec{x_i}; y_i\right) \quad (1)$$

in which a d-dimensional model $\vec{w}$ has to be found such that the objective function is minimized. The constants $\vec{x_i}$ and y_i, $1 \leq i \leq N$, correspond to the feature vector of the i[th] data example and its scalar label, respectively.

Gradient descent represents the most popular method to solve the class of optimization problems given in Eq. (1). Gradient descent is an iterative optimization algorithm that starts from an arbitrary model $\vec{w}^{(0)}$ and computes new models $\vec{w}^{(k+1)}$, such that the objective function, a.k.a., the loss, decreases at every step, i.e., $\Lambda(w^{(k+1)}) < \Lambda(w^{(k)})$. The new models $\vec{w}^{(k+1)}$ are determined by moving along the opposite gradient direction. Formally, the gradient $\nabla\Lambda(\vec{w}) = \left[\frac{\partial\Lambda(\vec{w})}{\partial w_1}, \ldots, \frac{\partial\Lambda(\vec{w})}{\partial w_d}\right]$ is a vector consisting of entries given by the partial derivative with respect to each dimension. The length of the move at a given iteration is known as the step size—denoted by $\alpha^{(k)}$. With these, we can write the recursive equation characterizing the gradient descent method:

$$\vec{w}^{(k+1)} = \vec{w}^{(k)} - \alpha^{(k)} \nabla\Lambda\left(\vec{w}^{(k)}\right) \quad (2)$$

To increase the number of steps taken in one iteration, stochastic gradient descent (SGD) estimates the Λ gradient from a subset of the training dataset. Notice that the model update is applied only to indices with non-zero gradient which correspond to the non-zero indices in the training example. In order for SGD to achieve convergence, the examples have to be processed in random order at each iteration. Parallelizing SGD is not straightforward because of a chain dependency on model updates, where the current gradient relies on the previous update. Two classes of parallel SGD algorithms stem from this problem. Model-merging SGD [43, 31] simply ignores the dependencies across data partitions and averages the partial models. HOGWILD! [28, 35] is a parallel lock-free SGD algorithm for shared-memory architectures that allows multiple threads to update a single shared model concurrently, without any synchronization primitives. The resulting non-determinism enhances randomness and guarantees convergence for sparse models.

[4] https://en.wikipedia.org/wiki/N-gram

In big model gradient descent optimization, the model $\vec{w}$ is too large to fit in memory. There is no assumption about the relationship between d and N—the number of examples can or cannot fit in memory. This makes gradient computation and model update considerably more complicated. In [32], the out-of-core dot-product $\vec{w} \cdot \vec{x}_i$ is identified as the primordial operation for big model training. In order to compute the out-of-core dot-product, the model $\vec{w}$ is range-based partitioned into pages on secondary storage. Each training example $\vec{x}_i$ accesses the model by requesting its corresponding pages. A two stage solution is proposed to optimize the dot-product computation. In the reordering stage, groups of examples $\vec{x}_i$ are partially reordered to minimize the number of secondary storage accesses to model $\vec{w}$. In the batch stage, the requests made to the same model partition are aggregated into a single mega-request. This reduces the number of calls to the model storage manager. However, the solution proposed in [32] has several limitations. First, the dot-product join operator is serial, with each dot-product $\vec{w} \cdot \vec{x}_i$ being computed sequentially. Second, range-based partitioning wastes memory since model entries are grouped together rather arbitrarily, based on their index. Finally, it is assumed that all the model pages accessed by an example $\vec{x}_i$ fit in memory. In this work, we address all these limitations and propose a scalable model and data-parallel SGD algorithm. While our focus is on disk-resident models, a similar problem exists between main memory and cache—how to optimize model access to cache? We let this topic for future work.

Key-value stores. Parameter Server [22] tackles the big model problem by partitioning the model $\vec{w}$ over the distributed memory of a cluster of "parameter servers". Essentially, the model is stored as (key, value) pairs inside a memory-resident distributed hash table, with the model index as key and the model value as value. This storage configuration allows the model to be accessed and modified by individual indices and avoids the drawback of wasting memory in range-based partitioning. Key-value representation is also more compact when storing sparse models considering that only the non-zero indices are materialized.

In this work, we focus on disk-resident key-value stores instead of distributed memory hash tables. Disk-resident key-value stores implement optimizations such as log-structured merge (LSM) trees. LSM trees [29, 3] are data structures optimized for random writes. They defer and batch disk writes, cascading the changes from memory to disk in an efficient manner reminiscent of merge sort. The scalable write performance of LSM-trees can be particularly useful for SGD since the model is updated frequently. Key-value stores also provide concurrent access to key-value pairs by applying sharding techniques on keys. Sharding splits the keys into different partitions in the buffer manager, allowing concurrent lock-free access to keys in distinct partitions. Key-value stores have a limited interface, where values are retrieved and updated using a simple `get/put` API. Even though `get/put` operations allow direct access to every index of the model $\vec{w}$, a large number of these operations have to be performed for big models. These can significantly amplify the overhead of the API function calls. In this paper, we study how to utilize the advantages of disk-based key-value stores for big models in order to reduce the number of `get/put` calls.

Vertical partitioning. Vertical partitioning [16, 42] is a physical design technique to partition a given logical relation into a set of physical tables that contain only a subset of the columns. One extreme is column-store, where each column of a table is stored as a partition. The other extreme is row-store, where there is only one partition containing all the columns. The purpose of vertical partitioning is to improve I/O performance for queries that access only a subset of the columns of a wide table by reading smaller vertical partitions instead of the full table. Vertical partitioning algorithms seek to find the optimal partitioning scheme for a given workload consisting of a set of `SQL` statements. The drawback of vertical partitioning is that the workload has to be known beforehand and extra cost is incurred when different vertical partitions are joined back to reconstruct the original table. There has been an abundance of work proposing vertical partitioning algorithms for different scenarios. The survey and detailed comparison in [16] is an excellent reference. In this work, we propose to use vertical partitioning techniques in order to improve the I/O performance of disk-based key-value stores by reducing the number of `get/put` requests made to the model $\vec{w}$. Moreover, different (key, value) pair partitions can be accessed concurrently.

3. HOGWILD! FOR BIG MODELS

In this section, we introduce the original shared-memory HOGWILD! algorithm [28] for stochastic gradient descent optimization and provide an immediate extension to big models. Then we identify the limitations of the straightforward algorithm and specify the challenges that have to be addressed to make it scalable.

Algorithm 1 HOGWILD!

1. **for** $i = 1$ **to** N **do** ***in parallel***
2. $\quad \vec{w}^{(k+1)} \leftarrow \vec{w}^{(k)} - \alpha^{(k)} \nabla \Lambda \left(\vec{w}^{(k)}, \vec{x}_{\eta^{(i)}}; y_{\eta^{(i)}} \right)$

HOGWILD! Conceptually, HOGWILD! is a very simple algorithm. It iterates over the examples $\vec{x}_i$ and applies the model update equation (2). However, this is done in parallel and the order in which examples are considered is random. Since there is no synchronization, the model at step $k+1$ can be written concurrently by multiple threads. This happens, though, only for the common non-zero entries—an example updates only its non-zero entries. As long as the examples are sparse relative to the number of features in the model, [28] proves that HOGWILD! achieves the same convergence as the serial implementation of SGD. Due to the complete lack of synchronization, it is expected that HOGWILD! achieves linear speedup. However, this is not the case for modern multi-core architectures because of the implicit cache-coherency mechanism that triggers the "pingpong" effect [35]. Fortunately, this effect can be eliminated by a simple modification to the algorithm—instead of updating model $\vec{w}$ for every example, we update it only once for a batch of examples. This technique is known as mini-batch SGD and requires each thread to make a local copy of the model $\vec{w}$.

Algorithm 2 Big Model HOGWILD!

1. **for** $i = 1$ **to** N **do** ***in parallel***
2. $\quad$ **for each** non-zero feature $j \in \{1, \ldots, d\}$ in $\vec{x}_{\eta^{(i)}}$ **do**
3. $\quad\quad$ `get` $\vec{w}^{(k)}[j]$
4. $\quad\quad$ compute $\nabla \Lambda \left(\vec{w}^{(k)}[j], \vec{x}_{\eta^{(i)}}; y_{\eta^{(i)}} \right)$
5. $\quad$ **end for**
6. $\quad \vec{w}^{(k+1)} \leftarrow \vec{w}^{(k)} - \alpha^{(k)} \nabla \Lambda \left(\vec{w}^{(k)}, \vec{x}_{\eta^{(i)}}; y_{\eta^{(i)}} \right)$
7. $\quad$ **for each** non-zero feature $j \in \{1, \ldots, d\}$ in $\vec{x}_{\eta^{(i)}}$ **do**
8. $\quad\quad$ `put` $\vec{w}^{(k+1)}[j]$
9. $\quad$ **end for**
10. **end for**

Naive HOGWILD! for big models. In the case of big models, the vector $\vec{w}$ cannot be fully stored in memory—not to mention replicated across threads. Since only portions of the model are

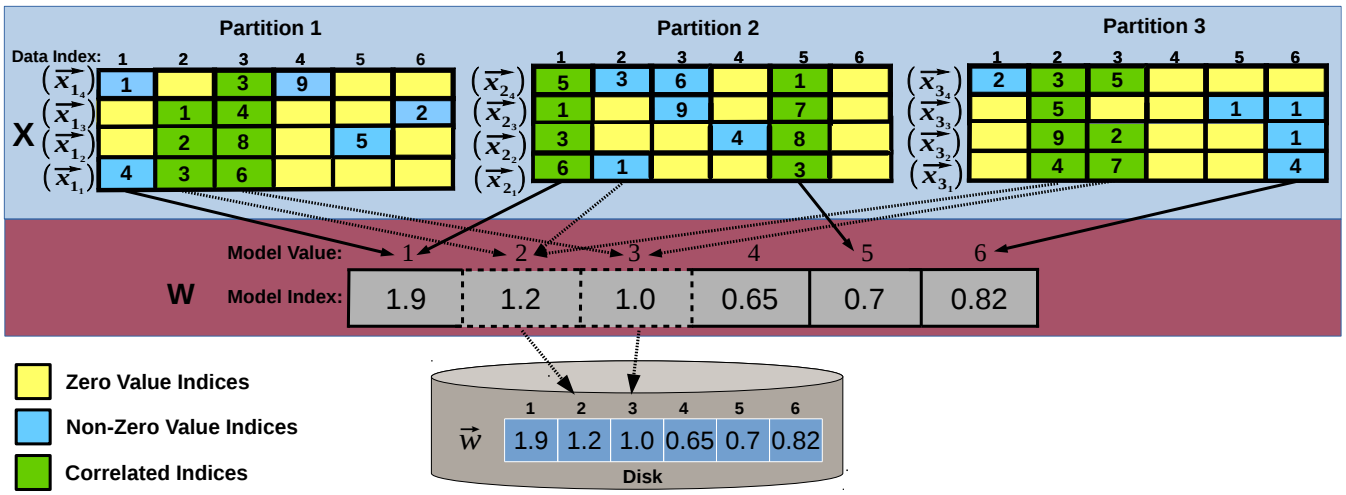

Figure 1: High-level approach in the proposed scalable HOGWILD! framework for big models.

cached in memory, every model access is potentially a disk access. Thus, model accessing becomes a series of requests to the storage manger—instead of simple memory references. This raises the question of how to store and retrieve the model from disk, i.e., what is the optimal model storage manager? Since the model is accessed at index granularity, a key-value store, e.g., LevelDB, makes for the perfect storage manager—the model index is the key and the model value corresponds to the payload. While in-memory key-value stores have been used before for model access due to their simple `get/put` interface [22, 5], as far as we know, this is the first attempt to use a disk key-value store for model management.

The immediate extension of HOGWILD! to big models – shown in Algorithm 2 – requires a `get` and a `put` call for each non-zero feature in an example, thus the explicit loops over the features. Given that there are many such non-zero features for every example, this puts tremendous pressure on the key-value store. The relatively random distribution of non-zero features across examples worsens the condition—to the point where the implicit sharding characteristic to key-value stores becomes irrelevant. Moreover, considering that half of the requests are `put`, the number of merge operations incurred by the LSM-tree is also high. While the number of `put` requests can be reduced with the mini-batch technique, this is hardly sufficient for big models because each `get` may require disk access. As with any other storage manager, key-value stores resort to caching in order to reduce the latency of `get/put` operations. Thus, they are susceptible to cache thrashing because the order in which `get/put` requests are issued matters. The naive HOGWILD! extension does not consider the request order inside a thread or across threads. Finally, the complete independence between threads in HOGWILD! becomes a limitation in the case of big models because model access is considerably more expensive than a memory reference. Copying a model index locally inside a thread becomes necessary—not an option specific to mini-batch SGD. This local copy provides an alternative for sharing that bypasses the key-value store and has the potential to eliminate the corresponding disk accesses from other threads.

4. SCALABLE HOGWILD!

We address the limitations of the naive HOGWILD! extension to big models by designing a novel model and data-parallel SGD framework specifically optimized for disk-based key-value stores. In this section, we provide a high-level overview of the proposed approach. The details are presented in subsequent sections.

Correlated indices. We illustrate the main idea of the proposed framework based on the example depicted in Figure 1. The sparse training examples $X = \{\vec{x}_{1_1}, \ldots, \vec{x}_{1_4}, \ldots, \vec{x}_{3_1}, \ldots \vec{x}_{3_4}\}$ are organized in 3 partitions as shown in the figure. The dense big model $\vec{w}$ is stored on disk with its working set W consisting of indices 1, 4, 5, and 6, respectively, kept in memory. It can be seen that partitions have correlated features, i.e., indices that co-occur across (almost) all the examples in the partition. For example, indices 2 and 3 in partitions 1 and 3 co-occur in 3 examples, while indices 1 and 5 co-occur in all the examples of partition 2. While index correlation is emphasized in Figure 1, this is a rather common characteristic of big models across analytics tasks in many domains—several real examples are provided in Section 1.

We design the scalable HOGWILD! framework by taking advantage of index correlation in both how the model is stored on disk and how it is concurrently accessed across threads. There are two stages in the scalable HOGWILD! framework. The offline stage aims to identify index correlations in the training dataset in order to generate a correlation-aware model partitioning that minimizes the number of `get/put` requests to the key-value store. In the online training stage, the model partitioning is used to maximize access sharing across threads in order to reduce the number of disk-level model accesses. We illustrate how each stage works based on the example in Figure 1.

Offline model partitioning. Existing solutions do not consider index correlation for storing and organizing the model on disk. They fall in two categories. In the first case, each index is assigned an individual key that is retrieved using the key-value store `get/put` interface [22]. This corresponds to columnar partitioning and incurs all the limitations of the naive HOGWILD! extension. Range-based partitioning is the other alternative [32]. In this case, indices that are adjacent are grouped together. For example, the range-based partitioning of model $\vec{w}$ constrained to two indices groups pairs $\{1, 2\}$, $\{3, 4\}$, and $\{5, 6\}$, respectively. However, this partitioning does not reflect the correlated indices in the training dataset, e.g., indices 2 and 3 still require two requests. An optimal partitioning of the model $\vec{w}$ groups indices $\{1, 5\}$, $\{2, 3\}$, and $\{4, 6\}$, so that indices 2 and 3 are concurrently accessed, effectively minimizing the number of requests to the model. We propose

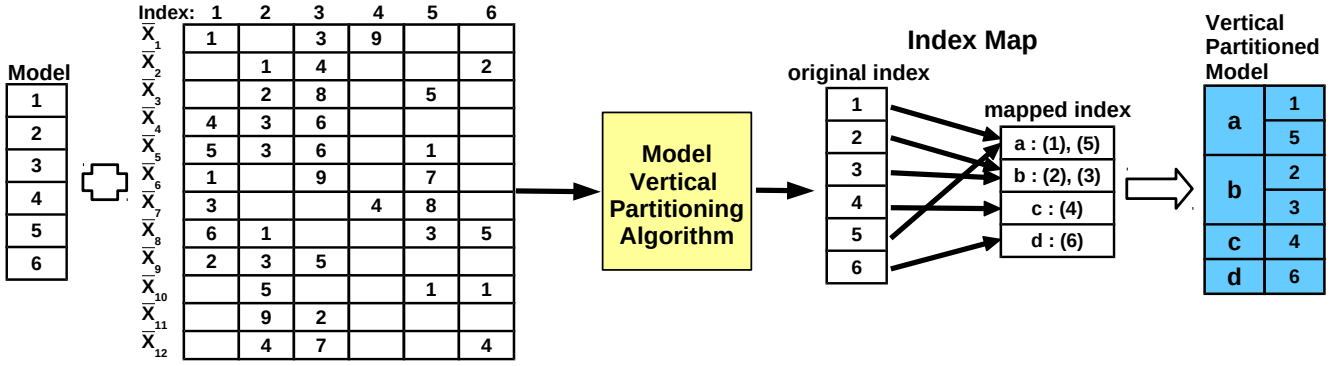

Figure 2: The offline model vertical partitioning stage.

a novel solution to find the optimal model partitioning by formalizing the problem as the well-known vertical partitioning [16] in storage access methods (Section 5).

Asynchronous online training. Data and model partitioning facilitate complete asynchronous processing in online training—the partitions over examples $\vec{x}_i$ are processed concurrently and themselves access the model partitions generated in the offline stage concurrently. This HOGWILD! processing paradigm – while optimal in shared-memory settings – is limited by the disk access latency in the case of big models. Model partitioning alone reduces the number of accesses for the correlated indices inside a training example—partition 2 incurs a single access to indices 1 and 5 for each example in Figure 1. However, it does not address correlated index accesses across examples and across threads. In Figure 1, we illustrate the case when data examples $\vec{x}_{1_1}$, $\vec{x}_{2_1}$, and $\vec{x}_{3_1}$ access the model $\vec{w}$ concurrently. The order in which requests are made plays an important role in enhancing cache locality. For example, if $\vec{x}_{1_1}$ and $\vec{x}_{2_1}$ request index 1 while $\vec{x}_{3_1}$ requests index 2, the opportunity of sharing access to index 2 – used by all the examples – is completely incidental. The same reasoning also applies to the examples inside a partition. At a first look, coordinating model access across partitions seems to require synchronization between the corresponding threads—exactly the opposite of the HOGWILD! approach. This is not the case. We devise an asynchronous solution that traverses the training examples in all the data partitions vertically, according to the model partitions generated offline. This enhances cache locality by allowing the same model index to be used by all the examples inside a partition. With this, partition 2 in Figure 1 incurs a single access to indices 1 and 5 for all the examples. In order to guarantee access sharing across partitions, we design an asynchronous mechanism in which partitions preemptively push the indices they acquire to all the other concurrent partitions. This is done only for `get` accesses. Following the example in Figure 1, when $\vec{x}_{3_1}$ obtains index 2, it pushes it to $\vec{x}_{1_1}$ and $\vec{x}_{2_1}$, effectively eliminating their request for index 2. The online stage is presented in Section 6.

Convergence considerations. We pay careful attention not to impact the convergence characteristics of the original HOGWILD! algorithm. Quite the opposite, the experimental results in Section 7 show that our use of correlation not only preserves the convergence of HOGWILD!, but improves it. This is in line with the results obtained in [40] where an orthogonal approach based on the concept of conflict graph is proposed. Abstractly, model partitioning is equivalent to the conflict graph—a partition corresponds to a connected component in the graph. Both are considered as a unit in model updates. As long as there is no interaction between partitions, i.e., connected components, [40] proves that a higher convergence rate than HOGWILD! can be achieved. When that is not the case, we get the default HOGWILD! behavior. We improve upon this by applying mini-batch updates. Instead of updating the model for each example – extremely inefficient because of the massive number of expensive `put` calls – we execute a single `put` for a batch. In order to avoid local model staleness [35], we precede the `put` with a `get` to obtain the latest model. This technique is shown to drastically improve convergence upon HOGWILD! [35]. To summarize, the proposed scalable HOGWILD! framework has the convergence guarantees given in [40] when partitions do not overlap and the HogBatch [35] behavior otherwise.

5. MODEL ACCESS MANAGEMENT

In this section, we present the offline model partitioning stage which serves to identify correlated indices in the training examples. The goal is to generate a correlation-aware partitioning that minimizes the number of model requests, i.e., `get/put` calls, for the entire dataset. This translates into a corresponding reduction in the number of disk accesses. To this end, we propose a composite storage scheme that maps correlated indices into a single key in the key-value store. We find the correlated indices by formalizing model access as vertical partitioning. We design a scalable frequency-based vertical partitioning algorithm able to cope with the dimensionality of big models and the massive number of training examples. The complete process is illustrated in Figure 2.

Model storage. In a straightforward implementation, each index and value in the model are stored as a key-value pair. For example, model index 2 in Figure 1 is represented as the key-value pair $(2 : 1.2)$, while model index 3 is mapped into $(3 : 1.0)$. Each of them is accessed independently with separate `get/put` calls. We introduce a novel composite storage scheme that considers index correlation and stores multiple indices under the same key. Following the example, since model indices 2 and 3 are correlated, the key-value pairs $(2 : 1.2)$ and $(3 : 1.0)$ are grouped together and stored as a composite payload under the new key $(a : (2 : 1.2), (3 : 1.0))$. The key feature of the composite storage scheme is that model indices are not stored individually, but clustered based on the correlations among them. This composite key-value mapping reduces the number of *get* requests to the key-value store, thus the number of disk seeks. The tradeoff is an increase in the payload size. We introduce a vertical partitioning algorithm that quantifies this tradeoff when deciding to merge two indices. Although we propose the composite scheme for disk-based key-value

stores, it may also be beneficial for in-memory hash tables since grouped indices are fetched to cache with a single instruction.

Model vertical partitioning. Given the set of training examples, the purpose of model partitioning is to identify the optimal composite key-value scheme. Specifically, we have to determine the number of keys and the payload corresponding to each key. The output of model partitioning is an index map that assigns correlated original indices to the same new index. For example, in Figure 2, index 2 co-occurs with index 3 seven times and index 1 co-occurs with index 5 four times. In the index map, index 1 and 5 are mapped to the same key a, while index 2 and 3 are mapped to key b. The remaining indices, 4 and 6, are mapped individually to c and d, respectively, even though index 6 co-occurs four times with index 2. The strategy to manage the index map – which can be extremely large for big models – is determined by the model partitioning algorithm. We can rewrite the set of training examples based on the computed index map as shown in Figure 3. Each training example $\vec{x}_i$ contains at most four keys, each of which possibly being a composite key. We emphasize that this rewriting is only logical—we do not create a new copy of the training data. The mapping is applied only during online training.

Our solution to model partitioning is inspired by vertical partitioning, where a similar problem is solved for optimal physical design at a much smaller scale—the number of columns in a table is rarely in the order of hundreds and the number of queries in the workload is rarely in the millions. Model partitioning can be mapped elegantly to vertical partitioning as follows. The training data correspond to the query workload, with each sparse example vector corresponding to a query and the non-zero indices to the accessed columns. The big model is equivalent to the logical relation that has to be partitioned. The difference is that the big model is essentially a one-row table that does not incur any join cost when partitioned. However, the cost of having small model partitions comes from the extra `get` requests made to the key-value store. As far as we know, we are the first to identify model partitioning as a vertical partitioning problem.

Vertical partitioning algorithm. We design a bottom-up model vertical partitioning algorithm [16, 42], depicted in Algorithm 3. It takes as input the model index set I, the sparse training examples X, and the cost function $cost(s)$ which computes the access cost for a partition of size s. It returns a model partitioning that minimizes the overall model access cost for the training set X. The algorithm uses a bottom-up greedy strategy inspired by [14]. The main loop is from line (2) to line (16). Initially, each model partition contains a single index. At each iteration, a pair of partitions that generate the largest reduction in the model access cost are merged together into a partition. This pair of partitions are identified by examining all the possible pairs of partitions in the current partitioning. The process repeats until no pair of partitions can be found to reduce the overall access cost.

Computing the reduction in cost of merging two partitions requires a pass over the training vector X. This is time-consuming when X contains a large number of examples—the case in analytics. Hill-Climb [14] alleviates this problem by pre-computing the cost for all $\mathcal{O}(2^d)$ partition pairs, where d is the dimensionality of the model. Considering the size of d for big models, this method is impractical. Instead of pre-computing all $\mathcal{O}(2^d)$ partitions, we compute an affinity matrix AF for the current partitioning at the beginning of each iteration (lines (3)–(6)). An entry $AF[i][j]$ in the affinity matrix represents how many times partition i co-occurs with partition j in the vector set X. Thus, the affinity matrix AF is symmetric, with the diagonal value $AF[i][i]$ representing how many times partition i appears. Using the affinity matrix, the subsequent computation of the merging cost can be performed without scanning the example set X for every candidate pair (lines (10)–(11)). This is an important optimization because of the size of X.

Algorithm 3 Model Vertical Partitioning

Require:
Model index set $I = \{1, 2, ..., d\}$
Sparse vector set $X = \{\vec{x}_1, \vec{x}_2, ..., \vec{x}_n\}$
Cost function $cost(s)$ for partition with size s
Ensure: Model partitions $P = \{P_1, P_2, ...\}$
1. **for each** index $i \in I$ **do** $P_i \leftarrow \{i\}$
Main loop
2. **while** true **do**
Compute affinity matrix AF for partitions in P
3. **for each** vector $\vec{x}_i \in X$ **do**
4. Collect partitions accessed by $\vec{x}_i$ in $P_{\vec{x}_i}$
5. Compute affinity $AF[i][j]$ for all pairs (P_i, P_j) in $P_{\vec{x}_i}$
6. **end for**
7. **for each** partition $P_i \in P$ **do**
8. Compute cost $C_i \leftarrow AF[i][i] \cdot cost(|P_i|)$
Select the best pair of partitions to merge
9. **for each** pair (P_i, P_j) in P **do**
10. Compute cost C_{ij} for partition $P_{ij} = P_i \cup P_j$:
$C_{ij} = (AF[i][i] + AF[j][j] - AF[i][j]) \cdot cost(|P_{ij}|)$
11. Compute reduction in cost if P_i and P_j are merged:
$\Delta_{ij} = C_i + C_j - C_{ij}$
12. **end for**
13. Pick the pair $(P_{i'}, P_{j'})$ with largest reduction in cost $\Delta_{i'j'}$
14. **if** $\Delta_{i'j'} = 0$ **return** partition P
15. **else** Merge $P_{i'}$ and $P_{j'}$
16. **end while**

Cost function. Assuming model indices are read from disk with no caching, the cost of reading a single index i is $c_i = t_s + t_u$, where t_s is the disk seek time and t_u is the time to read a one unit payload, i.e., one key-value pair. Assume also that only one disk seek is required per read. When indices i and j are grouped together under the same key, the cost of reading either of them is $c_i = t_s + 2 \cdot t_u$. Essentially, combining two indices generates a payload with twice the size, while the disk seek time stays the same. Given a partition P_i, the cost of accessing it is $cost(P_i) = t_s + |P_i| \cdot t_u$. This is the cost function used in the model vertical partitioning algorithm. Thus, the overall cost of accessing partition P_i across the set X is $AF[i][i] \cdot cost(|P_i|)$.

Sampling & frequency-based pruning. The time complexity of Algorithm 3 is $\mathcal{O}(Nd^2)$, where N is the number of examples in the vector set X and d is the dimensionality of the model. Given the scale of N and d, this complexity is infeasible for practical purposes. We propose a combined sampling and frequency-based pruning technique to reduce the number of examples and model indices considered by the algorithm. First, reservoir sampling [39] is applied to extract a sample of $N' \ll N$ examples. Second, frequency-based dimension pruning is executed over the samples. This involves computing the frequency of each model index occurring in the training data. Only the top-k most frequent indices are preserved and used in model partitioning. The non-extracted indices are kept as individual partitions. The intuition behind pruning is that infrequent indices have a reduced impact on reducing the partitioning cost. In the extreme case, an index that does not appear at all should not be considered for merging. Computing top-k over big models is challenging due to the size of the domain, i.e., the dimensionality of the model. Exact computation requires disk access to preserve the counts for all the indices. A key-value

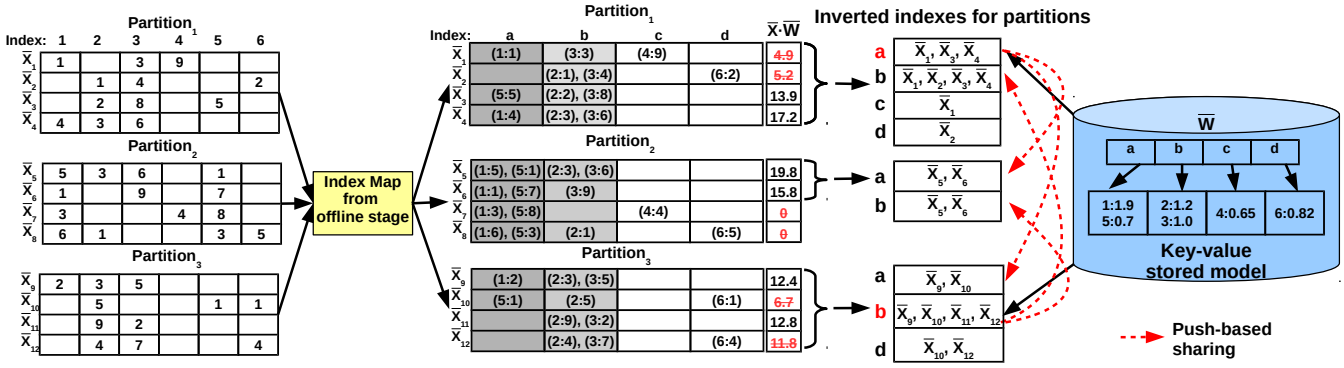

Figure 3: Online asynchronous training stage. Strikethrough dot-products are partial, while dashed indexes are pushed to other partitions.

store can be used for this purpose. Alternatively, approximate top-k methods based on sketches [34] have been shown to be accurate and efficient. This is what we use in our implementation.

6. ASYNCHRONOUS BIG MODEL SGD

In this section, we present how the scalable HOGWILD! framework performs asynchronous updates on vertically partitioned big models. Recall that the model indices are mapped to a smaller set of new keys (Figure 2). Two challenges have to be addressed. The first challenge is how to perform the SGD updates when the training data and the model have different representations. The index map built in the offline stage is used to translate between the two representations at runtime. The second major challenge is how to efficiently access the partitioned model such that disk requests are shared across partitions. We introduce a novel push-based mechanism to enforce sharing by vertical traversals of the example data and partial dot-product materialization.

On-the-fly data mapping. In order to accommodate the composite model storage scheme generated offline, we map the example vectors according to the index map. This is done on-the-fly, when the example is used in training. For example, in Figure 3, $\vec{x}_2$ in partition 1, which originally has three non-zero indices $(2:1)$, $(3:4)$, and $(6:2)$, respectively, is mapped into a vector with only two non-zero indices – $(b:(2:1),(3:4))$ and $(d:(6:2))$ – in which b is a composite key. Due to the correlation-aware model partitioning, the number of model requests made by example $\vec{x}_2$ is effectively reduced by 1 with data mapping. During parallel processing, data mapping is executed concurrently across all the active partitions. Since data mapping is a read-only operation to the index map, a single copy is shared across partitions.

Concurrent model access. Supporting the asynchronous HOGWILD! access – multiple data partitions request and update the model concurrently, without any synchronization – is considerably more difficult when the model is stored on disk than when it is in memory because the overhead of accessing distinct model indices differs dramatically. The overhead of accessing indices that are in the key-value store cache is considerably smaller than the overhead for indices on disk. We examine several alternatives to this problem. Batching and reordering have been proposed to improve disk-based model access locality in a serial scenario [32]. The idea is to buffer model requests and reorder them such that requests to the same model indices are grouped together in order to share model accesses. This technique not only reduces the number of model requests, but also improves cache locality. We can extend the technique to parallel settings by aggregating requests from multiple partitions and performing the serial algorithm. However, this introduces synchronization between partitions – fast partitions have to wait for slow partitions to finish their updates – and results in poor speedup—not much higher than the serial baseline. A more scalable strategy is to perform batching and reordering locally for each partition and allow the model requests to proceed concurrently without synchronization. Although this alternative provides higher concurrency, it suffers from poor cache locality since the indices requested by distinct partitions can be arbitrarily different.

Push-based model access sharing. We propose a novel push-based sharing technique that supports scalable asynchronous model access while achieving good cache locality. When a partition acquires a model index, it preemptively pushes the corresponding value to all the active partitions, before they start their own requests. This is realized with an inverted index data structure. Pushing achieves better cache locality since, otherwise, the model value may have been already evicted from the key-value store cache at the time other partitions are requesting it. Push-based sharing is an optimistic strategy that assumes the same model index is required by other partitions as well. When this is the case, it not only improves cache locality, but also saves other partitions from requesting the model, which increases contention to the key-value store. Reducing contention is a critical optimization in concurrent settings.

Inverted index. In order to further increase cache locality, a key-to-vector inverted index is built for each partition, before starting to request model indices. The inverted index is essentially a hash table having as key the model index and as value the example vectors that need the corresponding index. For example, in Figure 3, model index a is requested by examples $\vec{x}_1$, $\vec{x}_3$, and $\vec{x}_4$ in partition 1, shown in the first entry of the inverted index for partition 1. The inverted index guarantees that each model value required by a partition is requested only once. Partitions start to access model values only after they build their inverted index. Notice that preemptive model pushing happens concurrently with inverted index building, i.e., while a partition is building the inverted index some of the entries can be served by pushed model values from other partitions. Thus, highly-concurrent operations on the inverted index data structure are critical in achieving high speedup. We realize this with the concurrent cuckoo hash table [23] in our implementation.

We illustrate how the push-based model access strategy uses the inverted indexes based on the example in Figure 3. At the time instant shown in the figure, partitions 1, 2, and 3 have all finished data mapping and have started to access the model concurrently. Partitions 1 and 3 are ahead of partition 2 in that both of them have completely built their inverted indexes and started to request

model values from the key-value store, while partition 2 has built an inverted index only for examples $\vec{x}_5$ and $\vec{x}_6$. Assume that partition 1 is the first to request pair $(a : (1 : 1.9), (5 : 0.7))$ from the key-value store. It subsequently pushes this pair to partitions 2 and 3, both of which are requiring key a. After getting key a, partition 3 skips it and requests key b instead. Meanwhile, partition 2 continues building its inverted index. A similar action happens when partition 3 gets key b. The pair $(b : (2 : 1.2), (3 : 1.0))$ is pushed to partitions 1 and 2, saving partition 1 from requesting key b again. The dashed arrows in Figure 3 denote the model pushing directions. Notice that, even though partition 2 has not finished building its inverted index, examples $\vec{x}_5$ and $\vec{x}_6$ are still able to process indices a and b, respectively.

Asynchronous model updates. While preemptive pushing increases concurrency, maintaining multiple copies of the model for each partition is impractical given the scale of big models. We avoid this by pre-aggregating the pushed model values for each example vector. This is possible due to the nature of analytics tasks having as primitive computation the dot-product between an example vector and the model [32], i.e., $\vec{x}_i \cdot \vec{w}$. Dot-product is essentially a sum over the product of non-zero indices in $\vec{x}_i$ and $\vec{w}$. While [32] requires fetching in memory all the $\vec{w}$ indices corresponding to non-zero indices in $\vec{x}_i$, in this work, we support partial dot-product computation. Each example is attached a running dot-product that is incrementally computed as indices are acquired—or pushed by other partitions (Figure 3). Only when the dot-product is complete, the example can proceed to gradient computation. Instead of applying the gradient to the model for each example vector, we accumulate the gradients within one partition and execute batch model updates, i.e., mini-batch SGD. An inherent issue with mini-batch SGD is that each partition can read stale model values which do not reflect the updates from other partitions [35]. The staleness is more problematic for big models because processing a batch takes longer. In order to reduce staleness, we introduce an additional `get` call before the model is pushed to the key-value store. This `get` is only for the model indices updated inside the batch and guarantees that correlated indices co-located in the same composite key – but not accessed by the batch – are not overwritten with stale values. This minor complication is the consequence of co-locating correlated indices. From experiments, we observe that the additional `get` has a major impact on convergence, without increasing the execution time.

7. EXPERIMENTAL EVALUATION

In this section, we evaluate the convergence rate, scalability, and efficiency of the proposed HOGWILD! framework on four datasets – two synthetic and two real – for three popular analytics tasks—support vector machines (SVM), logistic regression (LR), and low-rank matrix factorization (LMF). We also study the isolated effect of model partitioning and push-based sharing on the number of disk accesses. We take as a baseline for comparison the HogBatch extension of the naive HOGWILD! algorithm which is shown to be considerably superior in practice [31, 35]. HogBatch buffers the gradient for a batch and executes a single model update, effectively reducing the number of `put` calls to one. Since this method is logically equivalent to our proposed solution, it allows for a more in-depth comparison for the same set of configuration parameters, e.g., partition size and step size. We denote HogBatch as KV (key-value) and the proposed method as KV-VP (key-value with vertical partitioning) throughout all the results. The serial dot-product join operator [32] corresponds to the KV-VP-1. We do not consider distributed solutions based on Hadoop, Spark, or Parameter Server because our focus is on single-node shared-memory parallelism. Specifically, the experiments answer the following questions:

- What is the effect of model partitioning and push-based sharing on the convergence rate and the number of disk accesses as a function of the degree of parallelism available in the system?
- How do vertical model partitioning and push-based sharing improve runtime and reduce disk access with respect to the baseline HOGWILD!?
- How scalable is the proposed solution with the dimensionality of the model, the degree of parallelism, and the key-value store cache size?
- What is the sensitivity of offline model partitioning with respect to the model dimensionality, the size of the training dataset, and the frequency-based pruning?
- How much overhead does the key-value store incur compared to an in-memory implementation for small models?

7.1 Setup

Implementation. We implement the proposed framework as User-Defined Aggregates (UDA) in a parallel database system with extensive support for executing external user code. We extend the HOGWILD! UDAs from Bismarck [11] with support for big models and implement the runtime optimizations presented in Section 6. The database supports multi-thread parallelism and takes care automatically of all the aspects related to data partitioning, UDA scheduling, and resource allocation. The user has to provide only the UDA code containing the model to be trained and the example data. The model is stored in HyperLevelDB [10]—an embedded key-value store forked from LevelDB[5] with improved support for concurrency. The UDAs access HyperLevelDB through the standard `get/put` interface. There is a single HyperLevelDB instance shared across the HOGWILD! UDAs. This instance stores the shared model and manages access across all the HOGWILD! UDAs. The inverted indexes are implemented using the highly-concurrent cuckoo hashmap data structure [23]. Each HOGWILD! UDA has an associated inverted index that is accessible to other HOGWILD! UDAs for push-based sharing. The offline model partitioning algorithm is also implemented using UDAs. A UDA computes approximate index frequencies using sketch synopses. These are fed into the UDA for vertical model partitioning. Essentially, our code is general enough to be executed by any database supporting UDAs.

System. We execute the experiments on a standard server running Ubuntu 14.04 SMP 64-bit with Linux kernel 3.13.0-43. The server has 2 AMD Opteron 6128 series 8-core processors – 16 cores – 28 GB of memory, and 1 TB 7200 RPM SAS hard-drive. Each processor has 12 MB L3 cache, while each core has 128 KB L1 and 512 KB L2 local caches. The average disk bandwidth is 120 MB/s.

Table 1: Datasets used in the experiments.

Dataset	#Dims	#Examples	Size	Model
`skewed`	1B	1M	4.5 GB	12 GB
`splice`	13M	500K	30 GB	156 MB
`matrix`	10Mx10K	300M	4.5 GB	80 GB
`MovieLens`	6Kx4K	1M	24 MB	120 MB

Methodology. We perform all the experiments at least 3 times and report the average value as the result. Each task is ran for 10 iterations and the reported time per iteration is the average value across the 10 iterations. The convergence rate is measured by the

[5] https://rawgit.com/google/leveldb/master/doc/index.html

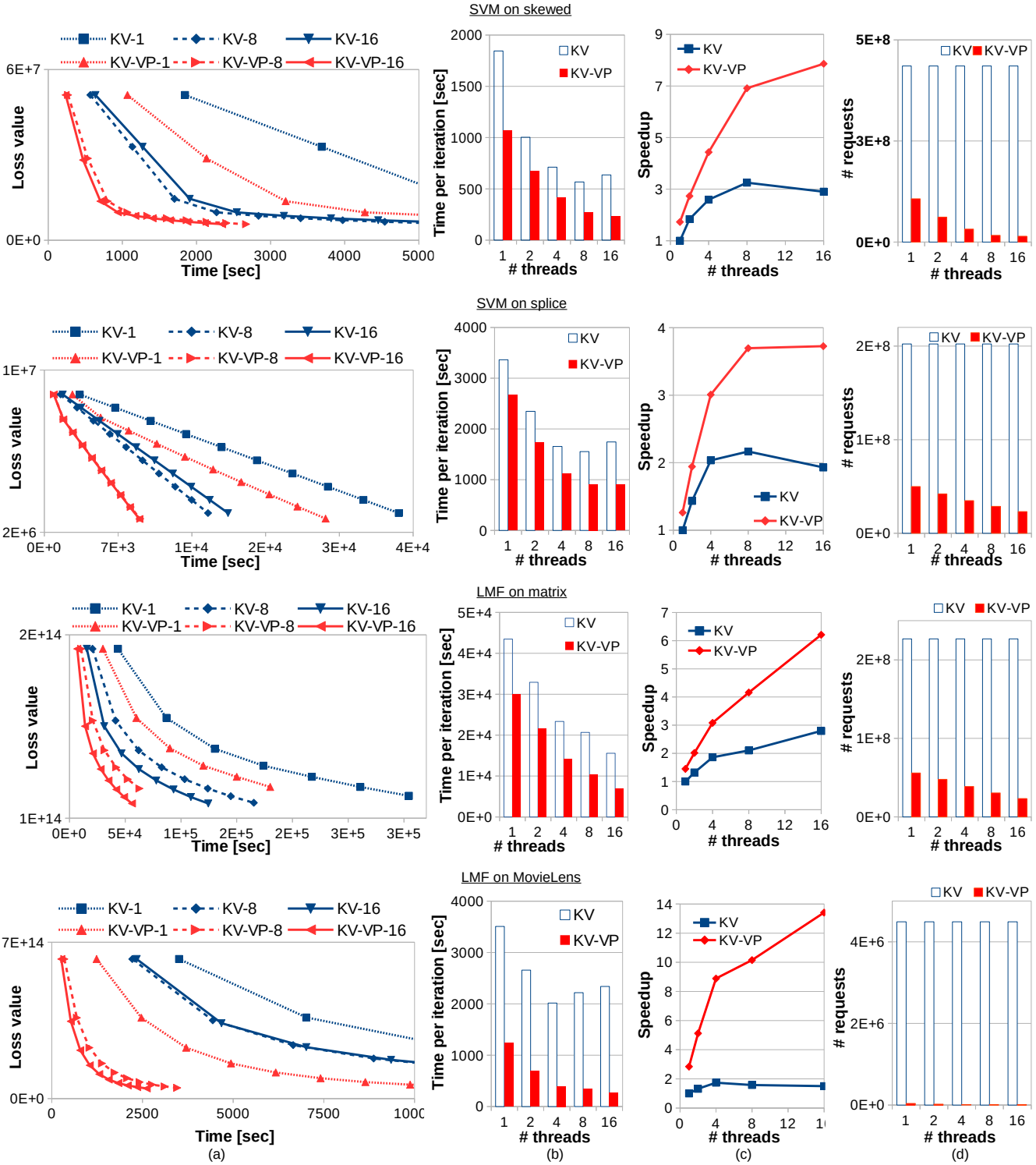

Figure 4: (a) Convergence over time. (b) Time per iteration. (c) Speedup over serial KV. (d) Number of key-value store requests.

objective function value at the end of each iteration. The time to evaluate the objective function value is not included in the iteration time. We always enforce data to be read from disk in the first iteration by cleaning the file system buffers before execution. Memory constraints are enforced by limiting the HyperLevelDB cache size to 1 GB across all the executions. Even with this hard restriction, HyperLevelDB uses considerably more memory – up to double the cache size – in the LSM-tree merging phase. This limitation guarantees that disk access is incurred for the large models used in the experiments. While the small models can be cached entirely by HyperLevelDB, in this case the results clearly show the benefit of reducing the number of model requests.

Measurements. We report the following measurements:

- *loss over time* represents the objective value as a function of the execution time. This measurement is taken for configurations with 1, 8, and 16 threads. The loss over time characterizes the convergence rate and how it changes with the number of threads. The purpose of the experiments is to identify the difference between the proposed method and HogBatch for a given set of configuration parameters, not to find the optimal hyper-parameters that give the best convergence rate.
- *speedup* is measured by dividing the execution time per iteration to the serial (1 thread) execution time for the baseline method. We do this for configurations with 1, 2, 4, 8, and 16 threads, respectively, and also report the *average time per iteration* separately. Speedup shows how scalable each method is.
- *number of requests* corresponds to the number of calls to the key-value store, while *number of reduced requests* to the number of requests saved by vertical partitioning.
- *model partitioning time* quantifies the overhead of the offline vertical partitioning stage.

Datasets and tasks. We run experiments over four datasets—two synthetic and two real. Table 1 depicts their characteristics. `skewed` contains sparse example vectors with dimensionality 1 billion having non-zero entries at random indices. The frequency of non-zero indices is extracted from a zipfian (zipf) distribution with coefficient 1.0. The index and the vectors are randomly generated. We do not enforce any index correlation in generating the example vectors. However, the high frequency indices are likely to be correlated. On average, there are 300 non-zero entries for each vector. The size of the model is 12 GB. `matrix` is generated following the same process, with the additional constraint that there is at least a non-zero entry for each index—if there is no rating for a movie/song, then it can be removed from the data altogether. `splice` [1] is a real openly-available dataset for distributed gradient descent optimization with much higher index frequency than `skewed`—close to uniform. `MovieLens` [11] is a small real dataset—both in terms of number of examples and dimensions. We emphasize that the model size, i.e., dimensionality, is the main performance driver and the optimizations proposed in this paper are targeting the model access efficiency. The number of examples, i.e., size, has only a linear impact on execution time. We execute SVM and LR over `skewed` and `splice` – we include only the results for SVM due to lack of space; the LR results are similar – and LMF with rank 1000 over `matrix` and `MovieLens`. We choose the dataset-task combination so that we cover a large set of configurations that illustrate the proposed optimizations.

7.2 Results

The results for online model training are depicted in Figure 4. We discuss the main findings for each measurement across all the experimental configurations to facilitate comparative analysis.

Convergence rate. The convergence rate over time of the baseline (KV) and the proposed scalable HOGWILD! (KV-VP) for three degrees of parallelism (1, 8, and 16) are depicted in the (a) subplots of the figure. KV-VP achieves similar – and even better – loss than KV at each iteration. This is also true for all the thread configurations and is especially clear for `splice`, which does not achieve convergence in the figure. Thus, parallelism does not seem to degrade convergence. The reason is our mini-batch implementation of SGD. While KV-VP converges faster as we increase the number of threads, this is not always true for KV – 16 threads are slower than 8 threads – due to a larger time per iteration. KV-VP always outperforms KV for the same number of threads. Even more, KV-VP-8 outperforms KV-16 in all the configurations. As a result, KV-VP achieves the same loss $2X$ faster for `splice` and `matrix`, $3X$ for `skewed`, and $7X$ faster for `MovieLens`. The gap is especially wide for `MovieLens`, where even the 1 thread KV-VP solution outperforms KV-16. This happens because LMF models exhibit massive index correlation that is successfully exploited by vertical model partitioning—we generate `matrix` without correlation. The 1 thread results isolate perfectly the effect of model partitioning since there is no sharing. When index correlation is high – the case for `skewed` and `MovieLens` – KV-VP significantly outperforms KV.

Time per iteration. The execution time per iteration is depicted in the (b) subplots of Figure 4. As the number of threads doubles, the execution time roughly halves. This is generally true up to 8 threads. At this point, the number of concurrent model requests overwhelms HyperLevelDB. The effect is considerably stronger for the baseline solution—the time goes up for 16 threads, except for `matrix`. Due to reducing the number of model requests through push-based sharing, KV-VP-16 still manages to improve upon KV-VP-8—at a lower rate, though. The isolated impact of vertical model partitioning is best illustrated by the 1 thread execution time. In this case, KV-VP is exactly the same as KV, plus model partitioning. The more index correlation in the training examples, the higher the impact of model partitioning. `skewed` and `MovieLens` clearly exhibit this with execution times that are as much as $3X$ faster for KV-VP than KV.

Speedup. The speedup corresponding to the time per iteration is shown in the (c) subplots of Figure 4. The reference is the execution time for KV-1. The speedup ranges between $4X$ for `splice` and $14X$ (almost linear) for `MovieLens`. In the case of LMF, the number of model requests is two times the rank, e.g., 2000. While KV has to execute this number of requests for each training example, model partitioning reduces the requests to 2 – in the best scenario – for KV-VP. The three orders of magnitude – probably less, on average – difference in the number of requests is the main reason for the (almost) linear `MovieLens` speedup. Push-based sharing is reflected in the KV-VP speedup over its serial instance KV-VP-1. This follows the same trend as the speedup over KV-1 and is in line with the execution time per iteration.

Number of key-value store requests. The effect of model access sharing across threads is depicted in the (d) subplots of Figure 4. Independent of the number of threads, KV issues the same number of requests. Sharing is only incidental—in the HyperLevelDB buffer manager. The number of requests in KV-VP decreases with increasing the number of threads. This shows that push-based sharing is able to reuse a model index across multiple threads. The reuse is larger for the correlated datasets `skewed` and `MovieLens`. The reduction in the number of requests between KV-1 and KV-VP-1 is entirely due to offline model vertical partitioning. It ranges between $4X$ and $100X$—for the highly-correlated `MovieLens` dataset.

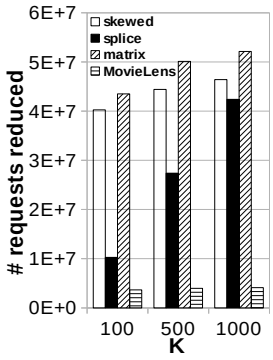

Figure 5: Key-value store requests eliminated by KV-VP.

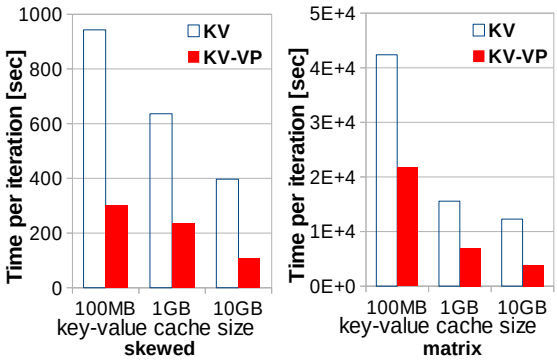

Figure 6: Key-value store cache size effect.

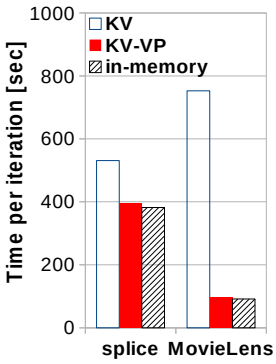

Figure 7: Key-value store overhead over in-memory.

Offline vertical partitioning impact on the number of key-value store requests. Figure 5 depicts the reduction in the number of model index requests for three values of K used in frequency-based pruning. The baseline is $K = 0$ which corresponds to no partitioning, i.e., each index is a separate key. The results show that the number of model requests is reduced by a very significant margin – as much as $4 \cdot 10^7$ – even when only the most frequent 100 indices are considered in partitioning. Thus, we use $K = 100$ to generate the model partitions for the experiments in Figure 4.

Key-value store cache size effect on big models. Figure 6 depicts the time per iteration as a function of the cache size for `skewed` and `matrix` – the big models in the experiments – with 16 threads. As expected, with the increase of the cache size, the execution time decreases because more data can be kept in memory. A small cache of 100 MB increases the KV execution time dramatically, especially for `matrix`. While a 10 GB cache provides improvement, this is relatively small—a factor of $2X$ or less.

Key-value store vs. in-memory implementation. We compare a memory-based key-value store with the HOGWILD! implementation in Bismarck [11] for the small models that fit in memory—`splice` and `MovieLens`. There is no difference in our code, except that the key-value store is mapped to *tmpfs*[6]—a memory-based file system. The goal is to identify the overhead incurred by the function calls to the key-value store. Figure 7 depicts the time per iteration for 16 threads. While the execution time of both KV and KV-VP improves when compared to the disk-based key-value store (Figure 4), KV-VP is hardly slower than the in-memory HOGWILD! This proves that storing the model in an optimized key-value store incurs minimal overhead – less than 5% – over a stand-alone solution.

Table 2: Vertical partitioning time (in seconds).

K	skewed	splice	matrix	MovieLens
100	722	79	52	55
500	2028	1404	224	181
1000	3054	4075	344	600

Vertical partitioning time. The time to perform the offline vertical model partitioning algorithm (Algorithm 3) is shown in Table 2. It is important to emphasize that this algorithm is executed only once for a given dataset. Moreover, this happens offline, not during training. Since Algorithm 3 is iterative, the execution time depends on the number of iterations performed – which are strongly dependent on the training dataset – and the number of samples N' considered by the algorithm. We observe experimentally that a 1% sample provides a good tradeoff between running time and partitioning quality. Table 2 shows that – as the pruning criteria is relaxed, i.e., K increases – the partitioning time increases. This is because the number of candidate partitions becomes larger. With the most restrictive pruning ($K = 100$), model partitioning takes roughly a KV-VP-16 iteration.

[6]https://en.wikipedia.org/wiki/Tmpfs

8. RELATED WORK

In-database analytics. There has been a sustained effort to add analytics functionality to traditional database servers over the past years. MADlib [15] is a library of analytics algorithms built on top of PostgreSQL. It implements the HOGWILD! algorithm for generalized linear models, such as LR and LMF, using the UDF-UDA extensions. Due to the strict PostgreSQL memory limitations, the size of the model – represented as the state of the UDA – cannot be larger than 1 GB. GLADE [31] follows a similar approach. Distributed learning frameworks such as MLlib [37] and Vowpal Wabbit [1] represent the model as a program variable and allow the user to fully manage its materialization. Thus, they cannot handle big models directly. The integration of relational join with gradient computation has been studied in [18, 19, 36]. In all these solutions, the model fits entirely in memory and they work exclusively for batch gradient descent (BGD), not SGD. The Gamma matrix [30] summarizes the training examples into a $d \times d$ matrix. While Gamma matrix provides significant size reductions when $d < N$, it cannot be applied in our setting where d is extremely large. Moreover, SGD does not benefit from the Gamma matrix because an update is applied for every example, while Gamma matrix summarizes all the examples across each dimension.

Big models. Parameter Server [21, 22] is the first system that addresses the big model analytics problem. Their approach is to partition the model across the distributed memory of multiple *parameter servers*—in charge of managing the model. In STRADS [20], parameter servers are driving the computation, not the workers. The servers select the subset of the model to be updated in an iteration by each worker. Instead of partitioning the model across machines, we use secondary storage. Minimizing the number of secondary storage accesses is the equivalent of minimizing network traffic in Parameter Server. The dot-product computation between training data and big models stored on disk is considered in [32]. Batching and reordering are proposed to improve disk-based model access locality in a serial scenario. Moreover, model updates require atomic dot-product computation. The proposed framework is both

model and data-parallel, does not reorder the examples, and supports incremental dot-product evaluation.

Stochastic gradient descent. SGD is the most popular optimization method used to train analytics models. HOGWILD! [28] implements SGD by performing model updates concurrently and asynchronously without locks. Due to this simplicity – and the near-linear speedup – HOGWILD! is widely used in many analytics tasks [33, 11, 25, 9, 8, 5]. HogBatch [35] provides an in-depth analysis of the asynchronous model updates in HOGWILD! and introduces an extension – mini-batch SGD – that is more scalable for cache-coherent architectures. This is our baseline for comparison. [40] partitions the training examples based on the conflict graph which corresponds to a model partitioning. This reduces model update conflicts across data partitions and results in higher speedup and faster convergence. While the proposed framework also applies model partitioning, the method and the target are different—reduce the number of model requests. Model averaging [43] is an alternative method to parallelize SGD that is applicable to distributed settings. Only HOGWILD! can be extended to big models. Averaging requires multiple model copies – one for each thread – that cannot be all cached in memory. A detailed experimental comparison of these methods is provided in [31].

9. CONCLUSIONS AND FUTURE WORK

In this paper, we present a general framework for parallelizing stochastic optimization algorithms over big models that cannot fit in memory. We extend the lock-free HOGWILD!-family of algorithms to disk-resident models by vertically partitioning the model offline and asynchronously updating the resulting partitions online. The experimental results prove the scalability and the superior performance of the framework over an optimized HOGWILD! extension to big models. In future work, we plan to explore two directions. First, we will investigate how the offline model partitioning can be integrated with online training. Second, we plan to explore how similar techniques can be applied when moving data from memory to cache.

Acknowledgments. This work is supported by a U.S. Department of Energy Early Career Award (DOE Career). We want to thank the anonymous reviewers for their insightful comments that improved the quality of the paper significantly.

10. REFERENCES

[1] A. Agarwal et al. A Reliable Effective Terascale Linear Learning System. *Machine Learning Research*, 15(1):1111–1133, 2014.
[2] Z. Cai et al. Simulation of Database-Valued Markov Chains using SimSQL. In *SIGMOD 2013*.
[3] F. Chang et al. Bigtable: A Distributed Storage System for Structured Data. *ACM TOCS*, 26(2):4, 2008.
[4] Y. Cheng, C. Qin, and F. Rusu. GLADE: Big Data Analytics Made Easy. In *SIGMOD 2012*.
[5] T. Chilimbi, Y. Suzue et al. Project Adam: Building an Efficient and Scalable Deep Learning Training System. In *OSDI 2014*.
[6] D. Crankshaw, P. Bailis et al. The Missing Piece in Complex Analytics: Low Latency, Scalable Model Management and Serving with Velox. *CoRR*, abs/1409.3809, 2014.
[7] A. Crotty et al. Tupleware: Redefining Modern Analytics. *CoRR*, abs/1406.6667, 2014.
[8] J. Dean et al. Large Scale Distributed Deep Networks. In *NIPS 2012*.
[9] J. Duchi, M. I. Jordan, and B. McMahan. Estimation, Optimization, and Parallelism When Data Is Sparse. In *NIPS 2013*.
[10] R. Escriva, B. Wong, and E. G. Sirer. HyperDex: A Distributed, Searchable Key-Value Store. In *SIGCOMM 2012*.
[11] X. Feng, A. Kumar, B. Recht, and C. Ré. Towards a Unified Architecture for in-RDBMS Analytics. In *SIGMOD 2012*.
[12] A. Ghoting et al. SystemML: Declarative Machine Learning on MapReduce. In *ICDE 2011*.
[13] M. Hall. Correlation-Based Feature Selection for Discrete and Numeric Class Machine Learning. In *ICML 2000*.
[14] R. Hankins and J. M. Patel. Data Morphing: An Adaptive, Cache-Conscious Storage Technique. In *VLDB 2003*.
[15] J. Hellerstein et al. The MADlib Analytics Library: Or MAD Skills, the SQL. *PVLDB*, 5(12):1700–1711, 2012.
[16] A. Jindal, E. Palatinus, V. Pavlov, and J. Dittrich. A Compararison of Knives for Bread Slicing. In *VLDB 2013*.
[17] A. Kumar, R. McCann et al. Model Selection Management Systems: The Next Frontier of Advanced Analytics. *SIGMOD Record*, 44(4):17–22, 2015.
[18] A. Kumar, J. Naughton, and J. M. Patel. Learning Generalized Linear Models Over Normalized Data. In *SIGMOD 2015*.
[19] A. Kumar, J. Naughton, J. M. Patel, and X. Zhu. To Join or Not to Join? Thinking Twice about Joins before Feature Selection. In *SIGMOD 2016*.
[20] S. Lee, J. K. Kim, X. Zheng, Q. Ho, G. A. Gibson, and E. P. Xing. On Model Parallelization and Scheduling Strategies for Distributed Machine Learning. In *NIPS 2015*.
[21] M. Li, L. Zhou, Z. Yang, A. Li, F. Xia, D. G. Andersen, and A. Smola. Parameter Server for Distributed Machine Learning. In *Big Learning NIPS Workshop 2013*.
[22] M. Li et al. Scaling Distributed Machine Learning with the Parameter Server. In *OSDI 2014*.
[23] X. Li and D. Andersen et al. Algorithmic Improvements for Fast Concurrent Cuckoo Hashing. In *EuroSys 2014*.
[24] H. Liu and L. Yu. Toward Integrating Feature Selection Algorithms for Classification and Clustering. *IEEE TKDE*, 17(4):491–502, 2005.
[25] J. Liu, S. Wright, V. Bittorf, and S. Sridhar. An Asynchronous Parallel Stochastic Coordinate Descent Algorithm. In *ICML 2014*.
[26] Y. Low et al. GraphLab: A New Parallel Framework for Machine Learning. In *UAI 2010*.
[27] B. McMahan et al. Ad Click Prediction: A View from the Trenches. In *KDD 2013*.
[28] F. Niu, B. Recht, C. Ré, and S. J. Wright. A Lock-Free Approach to Parallelizing Stochastic Gradient Descent. In *NIPS 2011*.
[29] P. O'Neil et al. The Log-Structured Merge-Tree (LSM-Tree). *Acta Informatica*, 33(4):351–385, 1996.
[30] C. Ordonez, Y. Zhang, and W. Cabrera. The Gamma Matrix to Summarize Dense and Sparse Data Sets for Big Data Analytics. *IEEE TKDE*, 28(7):1905–1918, 2016.
[31] C. Qin and F. Rusu. Speculative Approximations for Terascale Distributed Gradient Descent Optimization. In *DanaC 2015*.
[32] C. Qin and F. Rusu. Dot-Product Join: An Array-Relation Join Operator for Big Model Analytics. *CoRR*, abs/1602.08845, 2016.
[33] B. Recht, C. Ré, J. Tropp, and V. Bittorf. Factoring Non-Negative Matrices with Linear Programs. In *NIPS 2012*.
[34] P. Roy, A. Khan et al. Augmented Sketch: Faster and More Accurate Stream Processing. In *SIGMOD 2016*.
[35] S. Sallinen et al. High Performance Parallel Stochastic Gradient Descent in Shared Memory. In *IPDPS 2016*.
[36] M. Schleich, D. Olteanu, and R. Ciucanu. Learning Linear Regression Models over Factorized Joins. In *SIGMOD 2016*.
[37] E. Sparks et al. MLI: An API for Distributed Machine Learning. In *ICDM 2013*.
[38] A. Sujeeth et al. OptiML: An Implicitly Parallel Domain-Specific Language for Machine Learning. In *ICML 2011*.
[39] J. S. Vitter. Random Sampling with a Reservoir. *ACM TOMS*, 11(1):37–57, 1985.
[40] P. Xinghao, M. Lam et al. CYCLADES: Conflict-free Asynchronous Machine Learning. *CoRR*, abs/1605.09721, 2016.
[41] C. Zhang, A. Kumar, and C. Ré. Materialization Optimizations for Feature Selection Workloads. *ACM TODS*, 41(1):2, 2016.
[42] W. Zhao, Y. Cheng, and F. Rusu. Vertical Partitioning for Query Processing over Raw Data. In *SSDBM 2015*.
[43] M. Zinkevich, M. Weimer, A. Smola, and L. Li. Parallelized Stochastic Gradient Descent. In *NIPS 2010*.