ColumnSGD: A Column-oriented Framework for Distributed Stochastic Gradient Descent

¶§Zhipeng Zhang †Wentao Wu ‡Jiawei Jiang §Lele Yu ♯¶Bin Cui ‡Ce Zhang
¶*School of EECS & Key Laboratory of High Confidence Software Technologies (MOE), Peking University*
♯*Center for Data Science, Peking University & National Engineering Laboratory for Big Data Analysis and Applications*
†*Microsoft Research, Redmond, USA* ‡*ETH Zurich, Switzerland* §*Tencent Inc.*
¶{*zhangzhipeng, bin.cui*}*@pku.edu.cn,* †*wentao.wu@microsoft.com*
‡{*jiawei.jiang, ce.zhang*}*@inf.ethz.ch,* §*leleyu@tencent.com*

***Abstract*—Distributed machine learning (ML) has triggered tremendous research interest in recent years. Stochastic gradient descent (SGD) is one of the most popular algorithms for training ML models, and has been implemented in almost all distributed ML systems, such as Spark MLlib, Petuum, MXNet, and TensorFlow. However, current implementations often incur huge communication and memory overheads when it comes to *large models*. One important reason for this inefficiency is the *row-oriented* scheme (RowSGD) that existing systems use to partition the training data, which forces them to adopt a *centralized* model management strategy that leads to vast amount of data exchange over the network.**

We propose a novel, *column-oriented* scheme (ColumnSGD) that partitions training data by columns rather than by rows. As a result, ML model can be partitioned by columns as well, leading to a distributed configuration where individual data and model partitions can be *collocated* on the same machine. Following this locality property, we develop a simple yet powerful computation framework that significantly reduces communication overheads and memory footprints compared to RowSGD, for large-scale ML models such as generalized linear models (GLMs) and factorization machines (FMs). We implement ColumnSGD on top of Apache Spark, and study its performance both analytically and experimentally. Experimental results on both public and real-world datasets show that ColumnSGD is up to 930× faster than MLlib, 63× faster than Petuum, and 14× faster than MXNet.

I. Introduction

The increasing availability of large data [1], [2] has spawned intensive research and engineering efforts on developing *distributed* machine learning (ML) systems, such as Spark MLlib [3], Petuum [4], Angel [5], TensorFlow [6], and MXNet [7]. While the architectures and implementations of these systems vary, all of them employ stochastic gradient descent (SGD) [8] and its variants for training ML models.

Existing implementations of SGD adopt a *row-oriented* data partitioning scheme that induces a *centralized* model management strategy. For example, MLlib employs one master and multiple workers in its architecture. It partitions training data by rows and each worker owns one data shard. Meanwhile, the master is in charge of storing and maintaining the ML model. In each iteration of SGD, workers first pull the latest model from the master; they then compute gradients using the local data shard and send the computed gradients to the master. The master further aggregates the received gradients and updates the model accordingly. Other architectures follow a similar paradigm. For instance, systems that are based on parameter servers, such as Petuum and MXNet, conceptually just replace the single master by multiple servers to alleviate the apparent communication bottleneck on the master.[1]

While RowSGD systems work well on small models, their performance deteriorates when model size increases to *tens of millions or billions* [9]. There are two well-recognized reasons for this performance downgrade. First, memory consumption becomes a big issue under the single-master setting as the master has to store the entire model. Systems following the parameter-server (PS) architecture attempt to address this issue by further partitioning the model across the parameter servers. Second, communication overhead becomes prohibitive when gradients/models with billions of dimensions are transferred via network. Using multiple parameter servers (in lieu of a single master) can alleviate this, yet the total communication cost does not change — it is still proportional to the size of the model and is just redistributed over more machines.

In this paper, we propose ColumnSGD, a *column-oriented* framework for distributed SGD computation, with a focus on training large-scale ML models.[2] Unlike RowSGD systems, it partitions training data by columns (i.e., feature dimensions) rather than by rows. Moreover, ColumnSGD further partitions the *model* by columns, following the same column distribution used for data partitioning. This *collocation* of data and model enables a novel *distributed* model management strategy that completely avoids exchanging gradients and models over network. ColumnSGD also minimizes the memory consumption one could think of, by having *all* workers together store and maintain the model in parallel. Although column-oriented data partitioning strategies have been extensively studied [10]–[12], their applications in distributed SGD implementations have yet been explored, to the best of our knowledge.

In more detail, one can compute gradients of many ML models (e.g., GLMs) by performing element-wise operations using the model and feature vectors of data points, together with some "statistics" (e.g., inner product). In fact, we can break down this computation into two steps: (1) compute

[1]For ease of exposition, we use "RowSGD systems" to refer to these row-oriented solutions in the rest of this paper.

[2]The ML models in our study include, but not not limited to, generalized linear models (GLMs) and factorization machines (FMs).

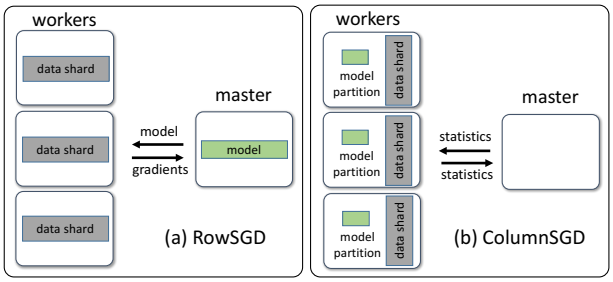


Fig. 1. Comparison of RowSGD and ColumnSGD.

statistics with the model and feature vectors, and (2) use the statistics to further compute gradient for each model/feature dimension. For example, the gradient of logistic regression can be expressed as a linear combination of the feature vectors (see Section II-C), and the statistics can be computed from the scalar dot products of the model vector and feature vectors. In RowSGD, one needs to pull the full model to compute the statistics. Our key insight that motivates the development of ColumnSGD is that the computation of those statistics can be decomposed and distributed in a column-oriented fashion. As a result, to compute the gradients one only needs to collect statistics rather than pull all dimensions of the model, which can dramatically reduce communication overhead as sizes of statistics are much smaller. (As an extreme example, for two vectors x_1 and x_2 both with one billion dimensions, their dot product $x_1 \cdot x_2$ remains as a single scalar!)

Figure 1 outlines the architecture of ColumnSGD and compares it with RowSGD, using a single-master setting. Similar to RowSGD, ColumnSGD employs one master and multiple workers. Column-partitioned training data and model are distributed across the workers. The master in ColumnSGD is thus lightweight. During each iteration of SGD, workers first compute aforementioned, *partial* statistics using their local data and model partition. They then send the statistics to the master. The master aggregates the partial statistics (e.g., when the statistics are in the form of dot products, the master simply adds up the partial dot products) and broadcasts back the complete statistics to the workers after aggregation. The workers finally use the received statistics to compute the gradients and update their corresponding model partitions.

The communication cost of ColumnSGD only depends on the *batch size* in SGD, whereas the communication cost of RowSGD depends on the *model size*. In practice, a batch size of 1,000 is usually sufficient to yield reasonable convergence of SGD (Section III). Therefore, for large models with millions of dimensions, the communication overhead of ColumnSGD is much lower than RowSGD. We have implemented a prototype system on top of Apache Spark. Our experimental evaluation on real-world ML workloads shows that ColumnSGD can be up to 930× faster than MLlib, 63× faster than Petuum, and 14× faster than MXNet, for large scale GLMs and FMs.

In summary, this paper makes the following contributions:

- We propose the ColumnSGD framework, a column-oriented implementation of distributed SGD for training large-scale ML models. It partitions both training data and model by columns, which enables a novel, distributed model management paradigm that is not allowed in RowSGD systems.
- We perform a detailed analysis of ColumnSGD using an analytic approach. ColumnSGD avoids sending gradients/models over network (as in RowSGD), which significantly reduces communication overhead for large models.
- To lower the additional overhead of converting row-oriented data to column-oriented one, we propose a protocol that performs block-level, column-wise data partitioning, as well as a two-phase indexing scheme that allows row-oriented mini-batch sampling on column-partitioned training data.
- We implement ColumnSGD on top of Apache Spark, and an extensive performance evaluation based on this implementation show that ColumnSGD can significantly outperform existing RowSGD systems on real-world ML workloads.

(Discussion) Just like all existing systems for distributed ML, ColumnSGD is clearly not a "one size fits all" type of solution. In particular, it is designed for workloads that require training large ML models using SGD, and we have found many use cases, including training large-scale GLMs and FMs, where ColumnSGD can significantly outperform other solutions. Our integration of ColumnSGD into the Apache Spark ecosystem further opens the door for Spark users to explore more use cases where ColumnSGD can be a suitable solution.[3]

(Paper Organization) We start by presenting necessary background in Section II. We then present the system architecture and programming framework of ColumnSGD, followed by an analytic comparison between ColumnSGD and RowSGD in Section III. In Section IV we further describe implementation details of ColumnSGD, and in Section V we report experimental evaluation results based on this implementation. We summarize related work in Section VI and conclude the paper in Section VII.

II. Parallelization Strategies in Distributed SGD

In this section, we first present the basics of SGD and its classic distributed implementation. We then discuss two parallelization strategies for computing gradients, namely, *horizontal-parallel* and *vertical-parallel*.

A. Stochastic Gradient Descent

Given a classification task with X representing the input data (a featurized vector representation), we look for a model w that minimizes the objective function:

$$f(w, X) = l(w, X) + \Omega(w). \quad (1)$$

Here, $l(w, X)$ is the *loss function*, which can be logistic loss, square loss, hinge loss, etc. $\Omega(w)$ is the regularization term that protects from overfitting (e.g., $\Omega(w) = \lambda|w|$).

Gradient descent is an algorithm that has been widely used to train ML models that optimizes Equation 1. In practice, people usually use a variant called stochastic gradient descent (SGD). We present the details of SGD in Algorithm 1.

[3]For ease of exposition, throughout this paper we will focus our presentation using GLMs as examples, whenever possible.

Algorithm 1: SGD $\{T,\ \eta\ w_0,\ X\}$

for *Iteration* $t = 1$ **to** T **do**
  Sample a batch of data X_B;
  Compute the gradient $g_t = \sum_{x_i \in X_B} \nabla l(x_i, w_{t-1})$;
  Update model as $w_t = w_{t-1} - \eta \cdot g_t - \eta \cdot \nabla\Omega(w_{t-1})$;

Here, T is the number of iterations, η is the learning rate, and w_0 is the initial model. As shown in Algorithm 1, SGD is an iterative procedure. In each iteration t, It performs the following steps: (1) Sample a batch of training data X_B; (2) Compute the gradient g_t of Equation 1 using X_B and the current model w_{t-1}; (3) Use g_t to update the model w_t.

B. Distributed SGD

The sequential version of SGD described in Algorithm 1 is not feasible when it comes to large models or distributed data storage. On this note, people have proposed various distributed versions of SGD. Next, we illustrate the idea of distributed SGD using the implementation in Spark MLlib as an example.

Algorithm 2: Distributed SGD $\{T,\ \eta\ w_0,\ X,\ K\}$

Master:
Initialize the model as w_0;
Issue loadData() to all workers;
for *Iteration* $t = 0$ **to** T **do**
  Issue computeGradients(t) to all workers;
  Aggregate gradients from workers: $g_t \leftarrow \sum_{k=1}^{m} g_t^k$;
  Update model: $w_t \leftarrow w_{t-1} - \eta \cdot (g_t) - \eta \cdot \nabla\Omega(w_{t-1})$;
Worker k = 1, ..., K:
Function *loadData():*
  Load a partition of data X_k;
Function *computeGradients(t):*
  Get model w_{t-1} from the master node;
  Sample a batch of training data X_{bk};
  Compute local gradient:
  $g_t^k \leftarrow \sum_{x_i \in X_{bk}} \nabla l(w_{t-1}, x_i)$;
  Send g_t^k to the master node;

In Spark MLlib, there is a master and K workers. The master is responsible for maintaining the model and scheduling the workers, whereas the workers are responsible for holding the data and computing the gradients. Algorithm 2 summarizes the procedure under this setting: (1) The master first partitions the data by row (i.e., each partition contains a set of data points) and each worker loads one partition; (2) Each worker gets the model from the master, and uses the local data and the received model to compute the gradients; (3) The master aggregates the gradients received from the workers, and uses the gradients to update the model. The steps (2) and (3) are repeated until convergence.

C. Parallel Computation of Gradients

As illustrated in Algorithm 2, parallel computation of gradients is one of the key steps in distributed SGD. We now look into it in more detail, using logistic regression (LR) as an example. When training LR, the gradient g over a batch of data X_B is

$$g(w, X_B) = \sum_{x_i \in X_B} \frac{-y_i}{1 + exp(y_i \cdot \sum_{j=1}^{m}(w_j \cdot x_{ij}))} \cdot x_i. \quad (2)$$

Here x_i and y_i are the feature vector and the label of the i^{th} data point, respectively. m is the dimension of the feature vector, and x_{ij} represents the j^{th} feature of x_i. For ease of exposition, we denote the first summation operator ($\sum_{x_i \in X_B}$) as S_h (for *horizontal* summation over the rows, i.e., data points), and denote the second summation operator ($\sum_{j=1}^{m}$) as S_v (for *vertical* summation over the columns, i.e., features).

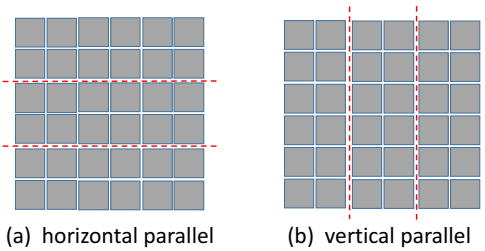


Fig. 2. Parallelization strategies of computing Equation 2. We represent the training dataset as a two-dimensional matrix, where each row represents a data point, and each column represents a feature. The *horizontal-parallel* strategy partitions the matrix by rows and distributes the computation of S_h, whereas the *vertical-parallel* strategy partitions the matrix by columns and distributes the computation of S_v.

(Horizontal-Parallel) Horizontal parallelization (*horizontal-parallel*) is the strategy illustrated in Algorithm 2) and has been widely used by existing distributed ML systems like Spark MLlib, Angel, Petuum, TensorFlow, and MXNet. When computing Equation 2, each worker computes a partial sum of S_h using data points (belonging to its local partition) with *all* their features (see Figure 2(a)).

(Vertical-Parallel) Vertical parallelization (*vertical-parallel*) instead distributes the computation of S_v: Each worker handles the computation over a vertical partition of the training data (see Figure 2(b)). Note that, under this scheme, the workers initially cannot compute the gradients because each of them only holds partial sums (corresponding to features in their local partitions) of S_v's. As we will discuss in Section III, the master needs to collect these partial sums from the workers, sum them up, and send the complete S_v's (for all features) back to the workers. Only until this point, the workers can proceed to compute the gradients. Nonetheless, under *vertical-parallel* each worker now has *all* data points in the training set. After receiving the complete S_v's from the master, they can compute the partial gradients and update the models without another trip to the master.

(Applicability) These two parallelization strategies can be used to compute gradients for many ML models. Notable examples include all GLMs, such as Least Squares, Logistic Regression (LR), Multinomial Logistic Regression (MLR), Support Vector Machine (SVM), as well as other nonlinear models such as Factorization Machine (FM) [13].

Algorithm 3: SGD in ColumnSGD $\{T,\ \eta\ w_0,\ X,\ m\}$

```
1  Master:
2  Issue initModel() to all workers;
3  Issue loadData() to all workers;
4  for Iteration t = 0 to T do
5      Issue computeStatistics() to all workers;
6      Issue reduceStatistics();
7      Broadcast the reduced statistics to all workers;
8      Issue updateModel() to all workers;
9  Function reduceStatistics():
10     Aggregate statistics from all workers;
11 Worker k = 1, ..., m:
12 Function loadData():
13     Load one vertical partition of the training data;
14 Function computeStatistics():
15     Sample a mini-batch of training data X_B;
16     Compute statistics using local partitions of training
         data and model;
17 Function updateModel():
18     Get statistics from the master;
19     Compute the gradients using the statistics and X_B;
20     Update the local model using the gradients;
```

III. The Column-SGD Framework

All existing systems for training GLMs use the *horizontal-parallel* strategy — the *vertical-parallel* strategy has yet been explored. In this section, we propose a novel programming framework called ColumnSGD based on *vertical-parallel*. In contrast, we refer to systems that are based on *horizontal-parallel* as RowSGD systems.

We first present the details of ColumnSGD and describe its SGD implementation. We then showcase how to train an LR classifier as an example. We further analyze the memory and communication overheads of ColumnSGD, and compare with RowSGD. Both theoretical and empirical evidences show that ColumnSGD is superior to RowSGD *for large models*.

A. SGD in ColumnSGD

In ColumnSGD, there is one master and multiple workers. We partition *both* the training data and the model by columns (using the same partitioning scheme), and each worker owns one single partition. Since data and model partitions are now "collocated" on each worker, we can then adopt the *vertical-parallel* strategy for computing gradients. ColumnSGD can thus avoid sending gradients through the network, which is expensive for large models with high dimensions. (Recall that vector representations of features, gradients, and models all bear the same number of dimensions.)

Algorithm 3 illustrates the execution of SGD. The master first asks the workers to load their columnar partitions of the training data and initialize their own partitions of the model (lines 2 and 3). The main loop of SGD starts afterwards. Each iteration includes the following steps:

- **Step 1** (lines 5, 14-16): Each worker computes "statistics" using its local partitions of the data and model. Different models may have different forms of statistics. For instance, statistics when training LR are in the form of dot products; they are in more complicated forms when training FMs.
- **Step 2** (lines 6 and 7): The master aggregates the statistics from all the workers and broadcasts the aggregated statistics back. The aggregation function is usually sum of the two inputs, though there is no restriction.
- **Step 3** (lines 8, 17-20): Each worker uses its local data and the statistics received from the master to compute the gradient and update its (local) model. The specific model update procedure depends on the variant of SGD in use (e.g., standard SGD, Adam, etc.).

(Remark) In Algorithm 3, by "statistics" we refer to the partial sums of S_v as in the LR example exhibited in Section II-C. As we have mentioned, this computation pattern is not tied to LR and can be used for other models such as MLR, SVM, and FM. Moreover, ColumnSGD can also work for variants of SGD such as Adam [14] and AdaGrad [15], by tweaking the implementation of model update in line 20.

1) Example – Logistic Regression (LR) in ColumnSGD and RowSGD: As a concrete example, we next showcase the implementation of LR in ColumnSGD, and compare with its implementation in RowSGD. Figure 3 presents detailed execution flows of both RowSGD and ColumnSGD. For simplicity, we assume that there is one master and two workers in both systems. The batch size is set to be four (i.e., in each iteration, each worker in RowSGD processes two data points). We now describe in detail what data is transported over the network and how the (partitioned) model is updated in both RowSGD and ColumnSGD.

(RowSGD) In RowSGD as shown in Figure 3(a), the master stores the model ($\vec{w}$) and each worker owns a horizontal partition of the training data ($\vec{x_1}$, $\vec{x_2}$ on worker$_1$ and $\vec{x_3}$, $\vec{x_4}$ on worker$_2$). The execution flow is as follows:

- Step 1. Each worker pulls the model ($\vec{w}$) from the master.
- Step 2. Each worker computes the gradient using the pulled model and the mini-batch data ($\vec{x_1}$, $\vec{x_2}$ for worker$_1$, $\vec{x_3}$, $\vec{x_4}$ for worker$_2$).
- Step 3. Each worker pushes the gradient ($\vec{g_1}$ for worker$_1$ and $\vec{g_2}$ for worker$_2$) to the master.
- Step 4. The master aggregates the gradients and updates the model ($\vec{w}$).

(ColumnSGD) In ColumnSGD as shown in Figure 3(b), each worker stores a vertical partition of the training data ($\vec{x_{11}}$, $\vec{x_{21}}$, $\vec{x_{31}}$, $\vec{x_{41}}$ for worker$_1$ and $\vec{x_{12}}$, $\vec{x_{22}}$, $\vec{x_{32}}$, $\vec{x_{42}}$ for worker$_2$) as well as the corresponding partition of the model ($\vec{w_1}$ for worker$_1$ and $\vec{w_2}$ for worker$_2$). The master does not need to store the model anymore. The execution flow is as follows:

- Step 1. Each worker computes the statistics using their local model partition and data partition. Here the statistics on each worker is a four-dimensional vector ($\vec{s_1}$ for worker$_1$ and $\vec{s_2}$ for worker$_2$).
- Step 2. Each worker pushes the statistics to the master.
- Step 3. The master sums up the statistics.

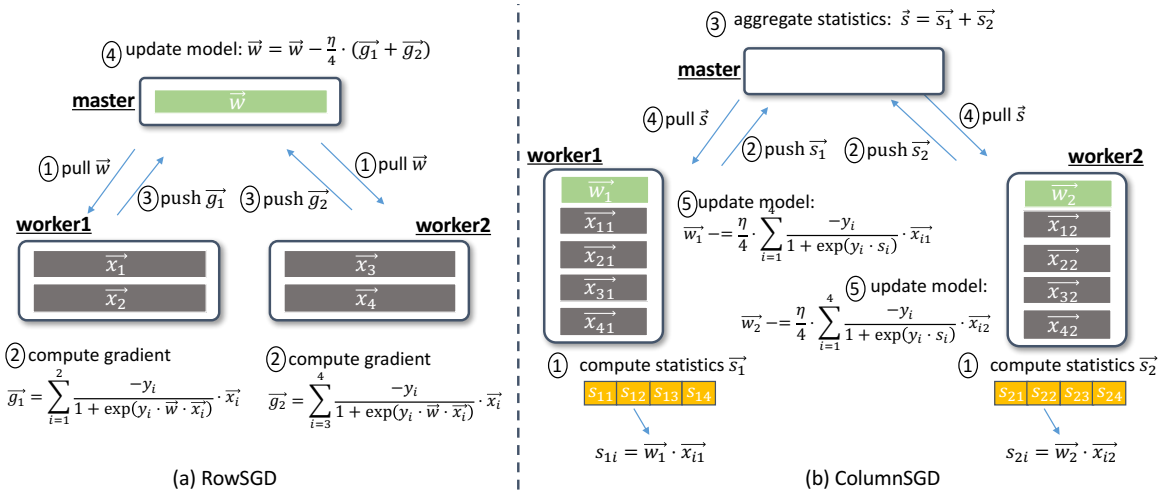


Fig. 3. Execution flows of Logistic Regression (LR) in RowSGD and ColumnSGD, using 1 master and 2 workers (with batch size 4 and learning rate η).

- Step 4. Each worker pulls the aggregated statistics ($\vec{s}$) from the master.
- Step 5. Each worker updates the model using the received statistics and their local data.

B. Analysis of ColumnSGD

We develop an analytic model to study the performance of ColumnSGD and further compare ColumnSGD with RowSGD to understand the pros and cons of both paradigms.

1) Memory and Communication Overheads: The computation costs of RowSGD and ColumnSGD are similar, because the computation is evenly distributed to the workers despite they adopt different parallelization strategies. Therefore, we focus on analyzing the memory and communication overheads. Again, we take training LR as an example. Analysis for other models and settings is similar.

We assume there is one master and K workers. The batch size of SGD is B. The dimension of the model is m and the *sparsity* (i.e., percentage of zeros) of the training data is ρ. The training data (including the labels) contains N points and the size of training data is $S = N + Nm(1-\rho)$.

(RowSGD) As depicted in Figure 1(a), in RowSGD the training data is partitioned by rows and each worker owns one partition. The master maintains the global model and schedules the workers. When running SGD, each worker needs to deal with $\frac{B}{K}$ data points. In expectation, there are $m\phi_1$ non-zero dimensions in a batch of $\frac{B}{K}$ data points, where $\phi_1 = 1 - \rho^{\frac{B}{K}}$. The master needs to aggregate the gradients from workers. In expectation, there are $m\phi_2$ non-zero dimensions in a batch of B data points, where $\phi_2 = 1 - \rho^B$.

Memory: The master needs to maintain the entire model and hold temporary memory for aggregating the gradients from workers. Therefore, the memory overhead for the master is $m + m\phi_2$. Each worker needs to store one horizontal partition of the training data as well as the temporary memory for the model parameters pulled from the master and the computed gradients. Thus the memory overhead is $\frac{S}{K} + 2m\phi_1$.

	RowSGD		ColumnSGD	
	master	worker	master	worker
Mem.	$m + m\phi_2$	$\frac{S}{K} + 2m\phi_1$	B	$\frac{S}{K} + 2B + \frac{m}{K}$
Comm.	$2Km\phi_1$	$2m\phi_1$	$2KB$	$2B$

TABLE I
MEMORY AND COMMUNICATION OVERHEADS.

Communication: Each worker needs to send the gradients computed using its local data batch and also pull back the model required by the local batch. Hence, the communication cost of each worker is $2m\phi_1$. On the other hand, the master sends messages to and receives messages from all workers. Hence, the communication cost of the master is $2Km\phi_1$.

(ColumnSGD) ColumnSGD partitions both the training data and the model by columns, as depicted in Figure 1(b). The master is responsible for aggregating and broadcasting only the statistics.

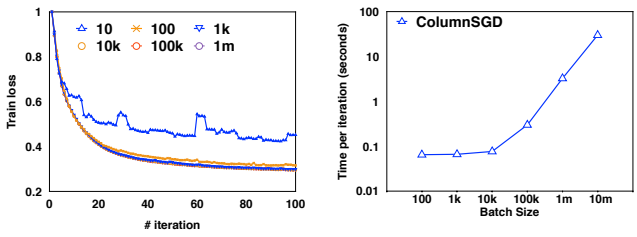
Memory: The master needs to hold the temporary memory for aggregating statistics, thus its memory overhead is B. On the other hand, each worker needs to store a partition of the training data as well as the model. Also it holds temporary memory for computing the statistics and pulling the aggregating statistics from the master. Thus the memory overhead is $\frac{S}{K} + 2B + \frac{m}{K}$.

Communication: The master aggregates statistics from workers and broadcasts the result back. The communication cost is $2B$ for each worker and $2KB$ for the master.

(Summary) As summarized in Table I, we conclude that:

- The memory overheads of the workers in RowSGD and ColumnSGD are similar, but the master in ColumnSGD is more lightweight as the model is offloaded to workers.
- The communication cost of ColumnSGD only depends on batch size whereas that of RowSGD depends on both model size, data sparsity and batch size.

2) Understanding Batch Size: While we find that the performance of ColumnSGD is more related to the batch size, it is unclear how the batch size would affect the convergence as well as the performance. It does not mean that we should always choose $B = 1$, as a larger batch size may result in faster convergence [16]. A natural question is then which batch size should be used.



(a) Loss w.r.t. #iterations (b) Per-iteration time

Fig. 4. SVM on kddb dataset with different batch size.

(Theoretical Understanding) As was pointed out by Yin et al. [17], the estimation of the gradient using a batch of training data can be modeled as the sum of the real gradient plus the "noise variance" brought by each data point. Then by applying central limit theorem [18], Yin et al. further proved that the noise variance is correlated with the *inverse* of the batch size. Therefore, to reduce the noise variance (and therefore speed up convergence of SGD), one may want to increase the batch size. Hence, there is indeed a sweet spot on the batch size that makes a tradeoff between communication overhead and convergence speed. We next study this tradeoff empirically.

(Empirical Study) We conduct an empirical study on the impact of batch size to understand the following two questions:

- How would batch size affect the convergence?
- How would batch size affect per-iteration overhead?

Specifically, we train LR and SVM using SGD on three datasets `avazu`, `kddb`, and `kdd12` (Table II) using the **Cluster 1** described in Section V. For each dataset, we first use grid search to find the best learning rate by *batch gradient descent* (i.e., we use the whole training data as one batch). We then fix the learning rate and keep *decreasing* the batch size. Figure 4(a) presents the convergence curve of training SVM on `kddb`. Results on `avazu` and `kdd12` are similar.

We have two observations. First, the convergence of SGD becomes unstable when the batch size is too small. For example, as shown in Figure 4(a), the convergence curve start to thrash when the batch size decreases to ten. The reason is that the variance brought by SGD increases as the batch size decreases, which is consistent with our analysis that the variance is inversely proportional to the batch size. Second, using too large batch size for linear models is not necessary. For example, the curves of SVM on `kddb` almost overlap when the batch size exceeds 100.

To further understand the impact of batch size on per-iteration overhead, we fix the learning rate and keep *increasing* the batch size when training SVM. Figure 4(b) presents the result on `kddb`. Results on the other datasets are similar.

We observe that the per-iteration time increases sharply when the batch size exceeds 100k. However, it remains almost unchanged for smaller batch sizes. When the batch size is small, the communication cost per iteration is dominated by the network latency. However, when the batch size is large, the communication cost is more affected by network bandwidth. In Figure 4(b), we observe near linear growth of communication time with respect to batch size beyond 100k, which is in line with the above analysis.

In summary, batch size can affect both convergence speed (i.e., the number of iterations to converge) and per-iteration time. Based on our empirical study, batch size of 1,000 can yield a good tradeoff for practical applications. This also indicates that small batch size is usually good enough, which we will focus on in the rest of the paper.

C. Discussion: ColumnSGD for Other Models

(Form of Statistics) ColumnSGD can work for many models besides LR (e.g., SVMs and FMs). Nevertheless, the types of "statistics" required by different models are different, which warrants customized implementations of statistics computations. As a result, the size of statistics may vary for different models, which can lead to different communication overhead and memory footprint. For example, implementation of FMs in ColumnSGD is more complicated than LR. For a batch of B data points, ColumnSGD needs to aggregate statistics of size $(F+1)B$ from each worker and broadcast $(F+1)B$ statistics to all workers. Here, F is number of latent factors. (We omit the derivation details due to space limitation.) Fortunately, F is typically a small number in practice. Therefore, the memory footprint of the statistics that ColumnSGD needs to keep for FMs remains small.

(Support for DNNs) ColumnSGD can support certain kinds of Deep Neural Networks (DNNs) but not all. For fully connected (FC) layers, ColumnSGD can support it by partitioning the FC layer and the corresponding weight matrix across workers. For `conv2d` or `maxpool` layers, however, it is difficult to partition the model by columns. Nevertheless, since the size of the model (i.e., the kernel size) is usually small in practice, synchronization is usually not expensive. Thus, one can just rely on RowSGD for `conv2d` and `maxpool` and we do not recommend ColumnSGD for such cases.

During the training process of DNNs with FC layers, ColumnSGD needs to synchronize every layer (both the forward pass and the backward pass). It needs to aggregate the dot products at each layer and broadcast the aggregated statistics (e.g., the result of activation functions) back to workers. However, given that the width of each individual layer in DNN is usually not large in practice, an implementation based on ColumnSGD may not be very beneficial.

IV. Implementation of Column-SGD

So far we have implicitly assumed that data and model have been partitioned by columns before ColumnSGD starts running SGD. However, in practice this assumption is often invalid, as training data stored in a distributed storage system such as HDFS [19] is often partitioned by rows. Moreover, stragglers are common in real world distributed ML jobs and we will present how to handle stragglers in ColumnSGD. We have implemented a prototype system on top of Apache Spark.

A. Row-to-Column Data Transformation

1) Design Desiderata: We present design considerations before implementation details.

(Row Identification) Although training data is partitioned by columns in ColumnSGD, it is still accessed by data points (i.e., rows) when taking a mini-batch in SGD. This introduces a constraint when designing the data transformation mechanism: Each partition must maintain information about the data points that the columns in this partition correspond to.

One idea could be to assign a global identifier to each row. However, it requires an additional full scan of all partitions, given that rows are scattered in multiple workers. To avoid this full scan, we instead use a composite key for each row that consists of the worker (and thus the row partition) identifier and the ordinal offset of the row within the partition. (We will further refine this row identification scheme using block-level identification discussed next.)

(Column Dispatching) Based on the previous scheme for row identification, a worker is now able to dispatch columns of its local rows to their destination workers. A straightforward dispatching scheme could be: (1) Each worker first loads its row partition and then splits it into K partitions by columns; (2) Each worker then dispatches its local column partitions to their designated owners.

This "bulk loading" scheme doubles the amount of required memory on each worker, which is not desirable. Instead, we consider an alternative that avoids increasing memory usage by partitioning each row "on the fly". Specifically, after loading a row, we partition it into K column groups and immediately send each group to its destination owner (We name this approach as "Naive-ColumnSGD".). However, naively doing this would significantly increase the communication overhead because there would be too many (small) objects to be transferred through network, in which the network bandwidth is under utilized. Therefore, we design a block-based (rather than row-based) dispatching scheme that essentially batches small objects before sending them out.

(Data Access) When using SGD to train ML models, we need to frequently sample a mini-batch of the training data. Some systems, such as Spark MLlib, simply perform a sequential scan over the data and decide one by one whether a data point should be included in the samples or not. This approach is clearly expensive for large training data. Other systems, such as TensorFlow, Petuum, and MXNet, adopt a different approach by partitioning the data into batches and reading a batch sequentially in every iteration. However, to ensure randomness, they have to shuffle the training data from time to time, which is also expensive in a distributed setting.

To speed up data access in ColumnSGD, we further design a two-phase indexing scheme based on the above block-level data transformation idea.

2) Implementation Details: We present the implementation details of row-to-column data transformation in ColumnSGD.

(Block-based Column Dispatching) Figure 5 illustrates the workflow of block-based column dispatching:

- **Step 1.** The master organizes the row-based training data into a queue of blocks, each with a predefined block size.
- **Step 2.** When a worker is idle, the master assigns one block to it by sending it a block ID. The worker then reads in the block, and splits it into K *worksets*. Each workset contains a column-based partition of the rows in this block as well as the block ID.
- **Step 3.** The worker then sends each workset to the destination worker together with the block ID. To further reduce the network traffic, we use the Compressed Sparse Row (CSR) format to represent each workset.

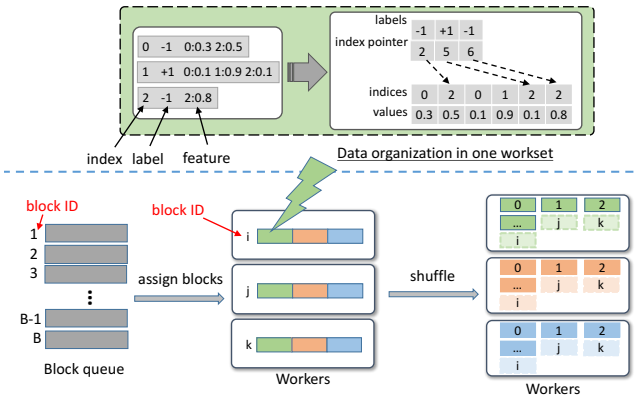


Fig. 5. Block-based column dispatching.

Algorithm 4: BlockBasedColumnDispatching(BQ, K)

```
1 foreach block b ∈ BlockQueue BQ do
2     Initialize K worksets (b.id, workset_1), ..., (b.id, workset_K);
3     foreach row r ∈ block b do
4         Assign each column of r to the corresponding workset using some predefined partitioning scheme (e.g., round robin);
5     for k ∈ {1, ..., K} do
6         Send (b.id, workset_k) to worker k;
7 Organize all worksets in each worker as a hash map;
```

Algorithm 4 presents a formal description of this procedure.

(Two-Phase Indexing) In ColumnSGD, we use a two-phase indexing scheme for accessing training data. The training data in each worker is organized as a hash map of received worksets (line 7 in Algorithm 4). The key to a workset is the ID of the block from which this workset comes. Inside each workset, a (partial) data point/row is further indexed by its offset.

When sampling a data point/row, each worker first draws a workset key using the same random seed (e.g., the current iteration number). This ensures that the workers can locate worksets from the same block simultaneously. Within that workset, each worker further draws an ordinal offset, again using the same random seed. This enables simultaneous landing on the same row in each worker.

B. Harnessing Stragglers

Stragglers have been recognized as one major performance issue in distributed computations, especially for systems based on the Bulk Synchronous Parallel (BSP) protocol [20]. Stragglers remain an important issue in ColumnSGD, as it naturally follows the BSP protocol as well.

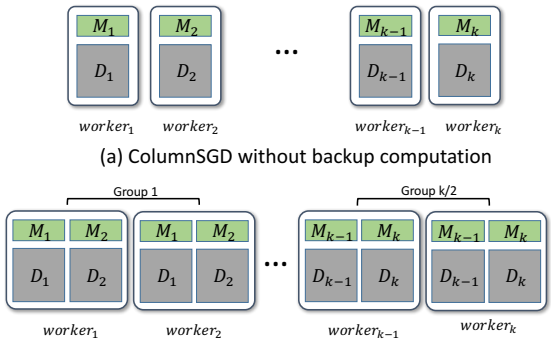


Fig. 6. Example of ColumnSGD w/o 1-backup computation. "M" stands for model partition and "D" for data partition.

Existing RowSGD systems, such as Petuum and MXNet, address stragglers by breaking synchronization barriers among workers when collecting gradients in distributed SGD [21], [22]. Can we leverage the same idea to handle stragglers in ColumnSGD? It turns out to be challenging, as the master has to collect the statistics from *all* workers. It is unclear whether ColumnSGD can use staled statistics (due to stragglers) to update the model without affecting the convergence of SGD.

To deal with stragglers in ColumnSGD, we turn to the recent development of *gradient coding theory* [23], [24]. The basic idea is to use *backup* (i.e., *redundant*) computation. In more detail, assume that we have K workers and the training data is partitioned into K parts. To ensure S-backup computation ($S << K$), we first divide the workers into $\frac{K}{S+1}$ groups and assign $(S+1)$ data partitions to each group disjointly. Each worker group handles the computation of statistics on the assigned $(S+1)$ data partitions. In each group, all workers are replicas of each other — each worker stores the $S+1$ data partitions as well as the corresponding model partitions. When running SGD in ColumnSGD with backup computation, each worker uses its local $(S+1)$ data partitions and model partitions to compute the statistics. The master then collects the statistics from non-straggler workers together with the corresponding worker ID. The master inspects the collected results until it can recover the correct statistics. It then kills those stragglers that have not finished the computation and broadcasts the aggregated statistics back to workers.

Figure 6 demonstrates a simple case where ColumnSGD is equipped with 1-backup computation. Figure 6(a) presents pure ColumnSGD, without backup computation. In this normal case, each worker takes care of its own data partition and the corresponding model partition. Figure 6(b) further demonstrates ColumnSGD with 1-backup computation. As we can see, the K workers are partitioned into $\frac{K}{2}$ groups and each group takes care of two partitions. For example, in the first group worker_1 and worker_2 serve as replicas of each other — each of them stores two data partitions (D_1 and D_2) and corresponding model partitions (M_1 and M_2). During training, each worker computes the aggregated statistics of all its partitions. If worker_1 becomes a straggler while worker_2 finishes in time, the master can recover the statistics for updating the model *without* the results from worker_1.

Dataset	#Instances	#Features	Dataset Size
avazu	40,428,967	1,000,000	7.4GB
kddb	19,264,097	29,890,095	4.8GB
kdd12	149,639,105	54,686,452	21GB
criteo	45,840,617	39	11GB
WX	69,581,214	51,121,518	130GB

TABLE II
DATASET STATISTICS.

To tolerate S stragglers, in theory we need S-backup computations, in which case both memory and computation costs increase by S times. However, the *communication* cost remains the same because it is only related to the batch size and the number of workers (Section III-B).

V. EXPERIMENTAL EVALUATION

We report experimental results in this section based on our implementation of ColumnSGD on top of Spark.

A. Experimental Setting

(Clusters) We use two clusters in our experiments:

- Cluster 1. This cluster consists of eight machines, each configured with 2 CPUs and 32 GB memory. The machines are connected via a 1Gbps network.
- Cluster 2. This cluster consists of 40 machines, each configured with 8 CPUs and 50 GB memory. The machines are connected via a 10 Gbps network.

We use **Cluster 1** for most of the experiments and use **Cluster 2** for scalability test.

(Datasets) All datasets used are publicly accessible[4] except that WX is from our industrial partner. We use (1) avazu, kddb and kdd12 mostly for evaluting baseline systems, and (2) criteo and WX for scalabilty test. Table II presents the statistics of the datasets.

(Workloads and Evaluation Metrics) First, we compare the time of data loading in ColumnSGD and MLlib since they are both implemented on top of Spark. Second, we compare the performance of training ML models on all baseline systems. Specifically, we train two GLMs (LR and SVM) using SGD on different datasets. We also train FMs [13], [25] to demonstrate the power of ColumnSGD because the model size of FM is considerably larger than GLMs. We report the training loss (i.e., the value of $f(w, X)$ in Equation 1) as time elapses. To further compare the performance of different systems, we also report the per-iteration time on these workloads.

(Baseline Systems) We compare ColumnSGD with four instances of RowSGD: (1) Spark MLlib 2.3.0 [3], an official ML library on top of Spark; (2) MLlib* [26], an optimized version of MLlib that combines model averaging with an AllReduce [27] implementation; and two PS implementations, i.e., (3) Petuum v1.1 [4] and (4) MXNet 1.3.0 [7]. Since TensorFlow does not have a mechanism for partitioning a single "variable", such as the weight matrix for LR, it is not suitable for training high dimensional models like GLMs and FMs [5], [28].[5] Therefore, we do not include TensorFlow.

[4]https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/,http://labs.criteo.com/2014/02/kaggle-display-advertising-challenge-dataset/

[5]https://www.tensorflow.org/guide/distribute_strategy

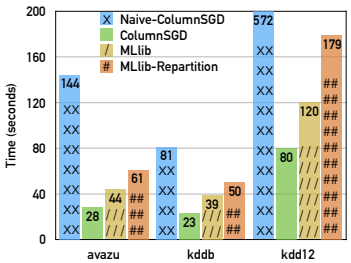


Fig. 7. Time cost of data loading.

(Parameter Settings) For each workload, we use grid search to tune the batch size and learning rate. For Spark MLlib, we further tune its configuration parameters such as the number of cores per executor, the number of partitions per core, etc. For PS-based systems, we set the number of servers same as that of workers. The batch size for all workloads in ColumnSGD, Petuum, MXNet and MLlib is 1,000 if not noted. We use the same hyper-parameter values for RowSGD and ColumnSGD (Table III), because they use the same optimization method (e.g., mini-batch gradient descent).

Dataset	LR	FM	SVM
avazu	10	10	1
kddb	10	10	1
kdd12	100	100	1
WX	0.1	0.1	0.01

TABLE III
LEARNING RATES OF BASELINE SYSTEMS ON DIFFERENT WORKLOADS.

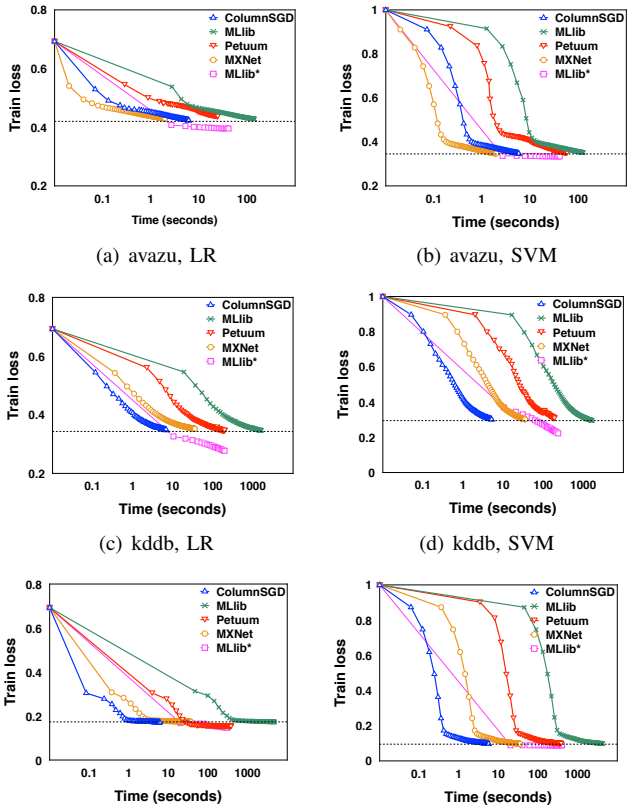
B. Evaluation of RowSGD and ColumnSGD

We compare RowSGD and ColumnSGD in this section. We first evaluate the performance of data loading (i.e., the row-to-column data transformation), then compare the convergence of all baseline systems.

1) Data loading: We first report the efficiency of data loading in ColumnSGD and MLlib. For ColumnSGD, we also include comparison with the "Naive-ColumnSGD" approach described in Section IV-A, which transforms the row-stored data into a column-partitioned one in a row-by-row fashion. Also, considering that we often incur a global data shuffling for load balance as well as statistical efficiency, we include MLlib for both with and without global data repartitioning. We assume that the training data is stored in HDFS by rows, which is the usual case. Figure 7 presents the time cost of data loading in different systems on three public datasets using **Cluster 1**. The result of MLlib* is not presented because MLlib* employs the same data loading mechanism as MLlib.

First, data loading in Naive-ColumnSGD is the slowest, for example it is $1.6\times\sim3.2\times$ slower than MLlib-Repartition and $2.1\times\sim4.7\times$ slower than MLlib. MLlib, MLlib-Repartition and Naive-ColumnSGD all process data points in a row-by-row, pipelined fashion. However, Naive-ColumnSGD needs to transfer $K\times$ more objects over network compared to MLlib, because it further partitions each data point into K pieces, where K is the number of workers. This leads to significant increase in serialization overhead before transferring data.

Second, data loading in ColumnSGD (i.e., block-based column dispatching) is the fastest. For example, it is $3.2\times\sim7.1\times$ faster than Naive-ColumnSGD on these three datasets. Furthermore, the performance of data loading in ColumnSGD is even faster than MLlib (without repartitioning) by $1.5\times\sim1.7\times$. Recall that, in ColumnSGD, we organize the training data in blocks and the network bandwidth is thus well utilized. Moreover, we further use CSR to compress the data blocks, which leads to much fewer objects to be serialized.



(a) avazu, LR (b) avazu, SVM

(c) kddb, LR (d) kddb, SVM

(e) kdd12, LR (f) kdd12, SVM

Fig. 8. ColumnSGD vs. RowSGD on LR and SVM.

2) Convergence of SGD: We present results for training GLMs and FMs separately.

(Results for LR and SVM) Figure 8 compares the convergence of training LR and SVM of all participating systems on the datasets avazu, kddb and kdd12.

We first compare the performance of ColumnSGD with MLlib, Petuum, and MXNet. Table IV reports the average per-iteration time of these systems when training LR. The results on training SVM are similar.

	MLlib	Petuum	MXNet	ColumnSGD	Speedup
avazu	1.43	0.24	0.02	0.06	**24/4/0.3**
kddb	16.33	1.96	0.3	0.06	**233/28/5**
kdd12	55.81	3.81	0.37	0.06	**930/63/6**

TABLE IV
PER-ITERATION TIME (SECONDS) OF TRAINING LR.

We observe that ColumnSGD gains significant speedup over MLlib and Petuum. The higher dimension the model has, the larger speedup ColumnSGD can achieve. For example, as shown in Figure 8(a), ColumnSGD is $24\times$ and $4\times$ faster than MLlib and Petuum on avazu. For a larger model on kdd12, ColumnSGD can achieve $930\times$ and $63\times$ speedup, respectively (Figure 8(e)). The reason for this trend is that the

communication cost of ColumnSGD only depends on the batch size. In contrast, the communication cost of MLlib and Petuum grows linearly with respect to the size of the model. We also observe that Petuum is an order of magnitude faster than MLlib. This is not surprising, though. Conceptually, Petuum uses multiple servers to replace the single node master in MLlib. The communication cost of the master is therefore evenly distributed across the servers.

On the other hand, ColumnSGD does not always outperform MXNet. For example, ColumnSGD is 5× and 6× faster than MXNet over `kddb` and `kdd12` when training LR (Figures 8(c) and 8(e)). However, it is 3× slower on `avazu` (Figure 8(a)). Inspecting Table IV, we find that the per-iteration time of MXNet increases as the model size increases, while that of ColumnSGD remains unchanged. This is again in line with our analysis that the communication time of ColumnSGD is only related to the batch size whereas that of MXNet depends on the model size. The observation that MXNet is faster than ColumnSGD on `avazu` is perhaps due to the scheduling latency in Spark. Meanwhile, MXNet is faster than MLlib and Petuum because it not only uses multiple servers to replace the master node, but also supports "sparse pull" of the model. That is, in each iteration MXNet only pulls the dimensions that are needed, whereas MLlib and Petuum have to pull all dimensions, which is apparently inefficient.

We then compare the performance of ColumnSGD against MLlib*. MLlib* [26] uses model averaging (MA) to improve both statistical and hardware efficiency. We observe the following: First, MLlib* can converge to a smaller training loss in some cases (e.g., on `kddb`) because MA can reduce variance [29]. Second, to achieve a certain loss (the horizontal line in each plot of Figure 8), ColumnSGD performs better on big models. For example, MLlib* converges faster than ColumnSGD on `avazu` but more slowly on `kdd12`.

(Results for FM) We now report the results on training FMs. FM models all interactions between features using factorized parameters and thus the model size is considerably larger than LR and SVM. A hyperparameter of FM is the number of factors used for each feature (we denote it as F here). We find that, with some mathematical derivation, the computation pattern of gradients in FMs can also be expressed as *vertical-parallel* and *horizontal-parallel*, except that the statistics are not as simple as "dot products" anymore.

We train FM on `avazu`, `kddb`, and `kdd12` by setting the factor F equal to ten. That is, the size of the model in FM is 10× larger than that of LR. To run an even larger model, we also set the factor to 50 on `kdd12`, which results in a model with more than 2.8 billion parameters (which is 21GB in FP64). Table V summarizes the results under these settings. Here, we do not compare ColumnSGD with MLlib, MLlib*, and Petuum because they do not implement FM.

Compared to MXNet, the speedup of ColumnSGD increases with the model size. For instance, ColumnSGD exhibits 14× speedup on `kdd12` when the factor F is set to ten. Since ColumnSGD sends statistics, rather than gradients and models as in MXNet, over the network, its communication overhead is much lower. Notably, ColumnSGD is able to handle model with more than 2.8 billion parameters while MXNet fails.

	MXNet	ColumnSGD	Speedup
avazu (F=10)	0.03	0.06	**0.5**
kddb (F=10)	0.56	0.06	**9**
kdd12 (F=10)	0.84	0.06	**14**
kdd12 (F=50)	OOM	0.15	**-**

TABLE V
PER-ITERATION TIME (SECONDS) OF TRAINING FM.

C. Dealing with Stragglers

We train LR using SGD on **Cluster 1** to demonstrate the efficacy of ColumnSGD using backup computation to deal with stragglers. For ease of illustration, we assume that there is only one straggler. To simulate the straggler, we randomly pick one worker in each iteration and let it sleep for some time according to *StragglerLevel*, which is defined as the ratio between the extra time a straggler needs to finish a task and the time that a non-straggler worker needs. For ColumnSGD with backup computation, each worker maintains two partitions of the training data as well as the corresponding model partitions. We compare the performance of (1) ColumnSGD with backup computation enabled (namely ColumnSGD-backup), (2) ColumnSGD with StragglerLevel set as 1 and 5 (namely ColumnSGD-SL1 and ColumnSGD-SL5, respectively). We also include the result of ColumnSGD without stragglers (namely ColumnSGD-pure) as a reference point. [6]

Figure 9 presents the per-iteration time for different settings. We observe that ColumnSGD without backup computation does suffer from stragglers. ColumnSGD-SL1 and ColumnSGD-SL5 are 2× and 6× slower than ColumnSGD-pure, respectively. In contrast, ColumnSGD with backup computation can avoid the effect of stragglers: The per-iteration time cost of ColumnSGD-backup is almost the same as that of ColumnSGD-pure.

D. Scalability Test

We test the scalability of ColumnSGD in terms of measuring its performance with respect to model size and cluster size.

(Scalability w.r.t. Model Size) We follow Christoph et al. [9], where they use the `criteo` dataset to generate synthetic datasets with different number of features. The synthetic datasets generated have model sizes varying from ten to one billion. However, the number of nonzero features remains stable regardless of the model size. We use SGD to train LR on these synthetic datasets on **Cluster 1** and Figure 10 presents the results. We observe that the per-iteration time of ColumnSGD remains stable when increasing the model size from ten to even one billion.

(Scalability w.r.t. Cluster Size) We train LR with SGD on `WX` dataset using {10, 20, 30, 40} machines in **Cluster 2**. Figure 11 presents the results. Specifically, Figure 11(a) characterizes the scalability of the data transformation procedure in

[6]To simulate the straggler in ColumnSGD-backup, we randomly pick one worker as the straggler. The straggler basically does nothing but returns null to the master. The rationale is the following. We can imagine that we first run an ML job for ten iterations and find that this worker is always slower. Then we just kill this worker and continue the training without data re-distribution.

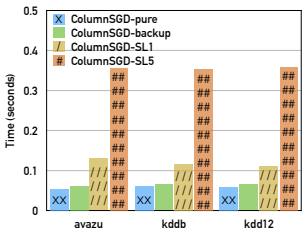


Fig. 9. Time with stragglers.

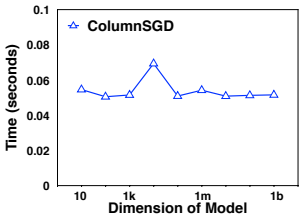
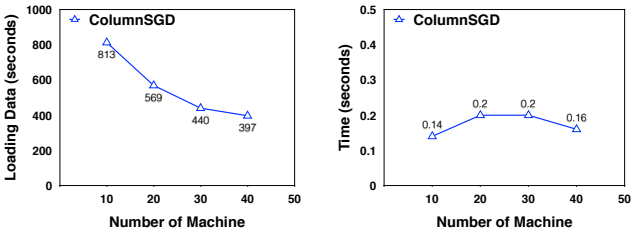


Fig. 10. Scalability w.r.t model size.



(a) Data Transformation Time (b) Per-Iteration Time

Fig. 11. Scalability w.r.t. cluster size.

ColumnSGD. As expected, the time on data loading decreases when the number of machines increases. For example, when we increase the number of machines from 10 to 20, we get 1.4× speedup. The speedup is somehow poor because we have to split each block and shuffle it among all workers. Overall, we observe a 2.05× speedup for transforming row-partitioned format to column-partitioned one when using 40 machines, compared to when ten machines are used.

On another note, per-iteration time remains almost unchanged when the number of machines increases, as shown in Figure 11(b). The per-iteration time cost is composed of computation cost and communication cost. As we know, when increasing the number of machines, the computation cost on each machine decreases, however, the communication cost increases. This indicates some potential scalability issues for ColumnSGD. It implies that, if we can fit the training data and model in memory, increasing the number of workers may not improve the latency of job training. This is a limitation of ColumnSGD that deserves future work.

VI. Related Work

(Column Partitioning in distributed ML) There exist many researches for column-oriented training data layout in distributed ML. DimmWitted [11] studies the tradeoff between different ML algorithms on the row- and column-oriented training data storage with a focus on NUMA-based system. DMac [30] explores both row-partitioning and column-partitioning for distributed matrix computation. Prasad et al. [31] integrates HP Vertica with distributed R, which allows both row- and column-oriented data access for different ML algorithms. Sibyl [12] explores column-oriented data layout for fast data access and data compression. There is also work that partitions the training data as well as the model into blocks by exploiting model-specific properties such as distributed matrix factorization [32] and topic modeling [33], [34]. Deep learning systems like SINGA [35] leverages model parallelism in training neural networks, which partitions the model vertically. However, each worker needs to store the whole dataset. Ordentlich et al. [36] also explored the idea of column partitioning for reducing the communication cost in `word2vec` model [37]–[39]. ColumnSGD is different in the following two aspects: (1) the data and model are *co-partitioned* by columns in ColumnSGD while this is not the case in [36]; (2) although both ColumnSGD and [36] reduce the communication cost by transferring some "statistics," the benefit is marginal for `word2vec` as the vector representation of each word contains several hundred dimensions at most, while for ColumnSGD the model size can be up to billions.

(Coordinate Descent) Unlike SGD, Coordinate Descent [40] (CD) is an optimization technique that naturally accesses training data in a column-oriented manner. Hydra [41] proposes a distributed CD algorithm where each worker updates different coordinates of the model. In contrast, in ColumnSGD we still access the train data in a row-oriented fashion. SDCA is a *coordinate ascent* algorithm that optimizes the dual problem. CoCoA [42] treats SDCA as a local solver, accelerates local computation in a primal-dual setting, and then combines partial results. However, CoCoA partitions the training data by row and needs to synchronize model updates among workers. mSDCA [43] further brings improvement to distributed SDCA with a mini-batch strategy. ColumnML [44] is a recent proposal that performs CD-based methods on columnar, in-database training data, targeting a CPU+FPGA platform.

(SGD in Existing ML Systems) State-of-the-art ML systems all partition the training data by rows when implementing SGD. In addition, data parallelism is widely adopted when training models like GLMs and FMs. Spark MLlib [3] follows a data-parallel paradigm. In MLlib, the master stores the model and each worker stores a row-based partition of the training data. When running SGD, each worker pulls the whole model from the master and computes the gradients. Another popular class of distributed ML systems leverage an architecture based on parameter servers, which conceptually simply uses multiple servers to replace the single-node master in Spark MLlib. The difference between these systems lies in how they maintain the model and how they pull the model. For example, TensorFlow [6] lacks mechanisms for model partitioning. In contrast, Petuum [4] and MXNet [7] can split a model into multiple pieces and store them on different servers. The major difference between Petuum and MXNet is that MXNet further supports "sparse pull," which allows for pulling a subset of model dimensions without retrieving the entire model.

(Stragglers in Distributed ML) There are two lines of work that aim for alleviating the impacts of stragglers in distributed ML training. One approach is to break the synchronization barrier in iterative ML training (e.g., [22]), where a worker may proceed without waiting for the slowest worker. However, this asynchronous approach breaks the serial consistency of distributed SGD and does not guarantee convergence. The other line is using backup computation (e.g., [23]), where coding theory is leveraged to create data block replica to avoid

the effect of stragglers. In ColumnSGD, we have followed the practice of the latter line of work. We replicate both data partition and model partition on each worker, such that the master node can recover the statistics following our predefined coding method when stragglers are detected.

VII. CONCLUSION

We have proposed ColumnSGD, a column-oriented framework for distributed SGD that targets training large-scale ML models. ColumnSGD partitions both training data and model by columns and enables a novel, distributed model management paradigm. Due to this collocation of data and model, ColumnSGD is able to significantly lower the communication overhead suffered by RowSGD systems when training large models, by avoiding sending gradients and models (with high dimensions) over the network. We presented the programming framework of ColumnSGD in detail, using logistic regression as a concrete example. We then performed an analytic comparison between RowSGD and ColumnSGD. We further provided an efficient row-to-column data transformation algorithm and solutions to handle stragglers in the context of ColumnSGD. We have also implemented ColumnSGD on top of Apache Spark and conducted an extensive experimental evaluation to demonstrate its effectiveness.

Acknowledgement This work is supported by the National Key Research and Development Program of China (No. 2018YFB1004403), NSFC(No. 61832001, 61702015, 61702016, 61572039), PKU-Tencent joint research Lab and Beijing Academy of Artificial Intelligence (BAAI).

REFERENCES

[1] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *HotCloud*, 2010.

[2] T. Zheng, G. Chen, X. Wang, C. Chen, X. Wang, and S. Luo, "Real-time intelligent big data processing: technology, platform, and applications," *Science China Information Sciences*, vol. 62, no. 8, p. 82101, Jul 2019.

[3] X. Meng *et al.*, "Mllib: Machine learning in apache spark," *CoRR*, vol. abs/1505.06807, 2015.

[4] E. P. Xing *et al.*, "Petuum: A new platform for distributed machine learning on big data," *IEEE Trans. Big Data*, vol. 1, no. 2, 2015.

[5] J. Jiang, L. Yu, J. Jiang, Y. Liu, and B. Cui, "Angel: a new large-scale machine learning system," *National Science Review*, vol. 5, no. 2, pp. 216–236, 2018.

[6] M. Abadi *et al.*, "Tensorflow: A system for large-scale machine learning," in *OSDI*, 2016, pp. 265–283.

[7] T. Chen *et al.*, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," *CoRR*, vol. abs/1512.01274, 2015.

[8] S. Ruder, "An overview of gradient descent optimization algorithms," *CoRR*, vol. abs/1609.04747, 2016.

[9] C. Boden, A. Spina, T. Rabl, and V. Markl, "Benchmarking data flow systems for scalable machine learning," in *SIGMOD*, 2017, pp. 5:1–5:10.

[10] D. J. Abadi, S. Madden, and N. Hachem, "Column-stores vs. row-stores: how different are they really?" in *SIGMOD*, 2008, pp. 967–980.

[11] C. Zhang and C. Ré, "Dimmwitted: A study of main-memory statistical analytics," *PVLDB*, vol. 7, no. 12, pp. 1283–1294, 2014.

[12] "Sibyl: A system for large scale supervised machine learning," https://users.soe.ucsc.edu/~niejiazhong/slides/chandra.pdf.

[13] S. Rendle, "Factorization machines," in *ICDM*, 2010, pp. 995–1000.

[14] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *CoRR*, vol. abs/1412.6980, 2014.

[15] J. C. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *JMLR*, vol. 12, pp. 2121–2159, 2011.

[16] L. Bottou, F. E. Curtis, and J. Nocedal, "Optimization methods for large-scale machine learning," *CoRR*, vol. abs/1606.04838, 2016.

[17] P. Yin, P. Luo, and T. Nakamura, "Small batch or large batch?: Gaussian walk with rebound can teach," in *SIGKDD*, 2017, pp. 1275–1284.

[18] O. Johnson, "Information theory and the central limit theorem," 01 2004.

[19] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *MSST*, 2010, pp. 1–10.

[20] G. Ananthanarayanan, S. Kandula, A. G. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris, "Reining in the outliers in map-reduce clusters using mantri," in *OSDI*, 2010, pp. 265–278.

[21] H. Cui, J. Cipar, Q. Ho, J. K. Kim, S. Lee, A. Kumar, J. Wei, W. Dai, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing, "Exploiting bounded staleness to speed up big data analytics," in *USENIX ATC*, 2014, pp. 37–48.

[22] E. P. Xing, Q. Ho, P. Xie, and W. Dai, "Strategies and principles of distributed machine learning on big data," *CoRR*, vol. abs/1512.09295, 2015.

[23] K. Lee, M. Lam, R. Pedarsani, D. S. Papailiopoulos, and K. Ramchandran, "Speeding up distributed machine learning using codes," in *ISIT*, 2016, pp. 1143–1147.

[24] R. Tandon, Q. Lei, A. G. Dimakis, and N. Karampatziakis, "Gradient coding: Avoiding stragglers in distributed learning," in *ICML*, 2017, pp. 3368–3376.

[25] S. Rendle, "Scaling factorization machines to relational data," *PVLDB*, vol. 6, no. 5, pp. 337–348, 2013.

[26] Z. Zhang, J. Jiang, W. Wu, C. Zhang, L. Yu, and B. Cui, "Mllib*: Fast training of glms using spark mllib," in *ICDE*, 2019, pp. 1778–1789.

[27] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of collective communication operations in MPICH," *IJHPCA*, vol. 19, no. 1, pp. 49–66, 2005.

[28] K. Zhang, S. Alqahtani, and M. Demirbas, "A comparison of distributed machine learning platforms," in *ICCCN*, 2017, pp. 1–9.

[29] J. Zhang, C. D. Sa, I. Mitliagkas, and C. Ré, "Parallel SGD: when does averaging help?" *CoRR*, vol. abs/1606.07365, 2016.

[30] L. Yu, Y. Shao, and B. Cui, "Exploiting matrix dependency for efficient distributed matrix computation," in *SIGMOD*, 2015, pp. 93–105.

[31] S. Prasad, A. Fard, V. Gupta, J. Martinez, J. LeFevre, V. Xu, M. Hsu, and I. Roy, "Large-scale predictive analytics in vertica: Fast data transfer, distributed model creation, and in-database prediction," in *SIGMOD*, 2015, pp. 1657–1668.

[32] R. Gemulla, E. Nijkamp, P. J. Haas, and Y. Sismanis, "Large-scale matrix factorization with distributed stochastic gradient descent," in *SIGKDD*, 2011, pp. 69–77.

[33] L. Yu, B. Cui, C. Zhang, and Y. Shao, "Lda*: A robust and large-scale topic modeling system," *PVLDB*, vol. 10, no. 11, pp. 1406–1417, 2017.

[34] A. J. Smola and S. M. Narayanamurthy, "An architecture for parallel topic models," *PVLDB*, vol. 3, no. 1, pp. 703–710, 2010.

[35] W. Wang *et al.*, "Singa: Putting deep learning in the hands of multimedia users," in *ACM Multimedia*, 2015, pp. 25–34.

[36] E. Ordentlich *et al.*, "Network-efficient distributed word2vec training system for large vocabularies," in *CIKM*, 2016, pp. 1139–1148.

[37] B. Li, A. Drozd, Y. Guo, T. Liu, S. Matsuoka, and X. Du, "Scaling word2vec on big corpus," *Data Science and Engineering*, vol. 4, no. 2, pp. 157–175, Jun 2019.

[38] S. Bonner, I. Kureshi, J. Brennan, G. Theodoropoulos, A. S. McGough, and B. Obara, "Exploring the semantic content of unsupervised graph embeddings: An empirical study," *Data Science and Engineering*, vol. 4, no. 3, pp. 269–289, Sep 2019.

[39] Q. Zhang, R. Li, and T. Chu, "Kernel semi-supervised graph embedding model for multimodal and mixmodal data," *Science China Information Sciences*, vol. 63, no. 1, p. 119204, Oct 2019.

[40] S. Shalev-Shwartz and A. Tewari, "Stochastic methods for l_1-regularized loss minimization," *Journal of Machine Learning Research*, vol. 12, pp. 1865–1892, 2011.

[41] P. Richtárik and M. Takáč, "Distributed coordinate descent method for learning with big data," *JMLR*, vol. 17, no. 1, pp. 2657–2681, 2016.

[42] M. Jaggi, V. Smith, M. Takác, J. Terhorst, S. Krishnan, T. Hofmann, and M. I. Jordan, "Communication-efficient distributed dual coordinate ascent," in *NIPS*, 2014, pp. 3068–3076.

[43] M. Takác, P. Richtárik, and N. Srebro, "Distributed mini-batch SDCA," *CoRR*, vol. abs/1507.08322, 2015.

[44] K. Kaan, E. Ken, Z. Ce, and A. Gustavo, "Columnml: Column-store machine learning with on-the-fly data transformation," *PVLDB*, pp. 348–361, 2019.

VIII. Supported Models

In this section, we present the implementation of various ML models using ColumnSGD. For each model, we first present the loss function it leverages and how to compute the gradients given the loss function. We then illustrate how to compute the partial statistics on each worker, how to aggregate the statistics on the master, and how to use the aggregated statistics to recover the gradients. We use the following notations in our following presentation:

- $X = \{(x_i, y_i), i \in [N]\}$, the training data that contains N data points;
- w, the model parameter;
- m, the number of the features;
- $l(\cdot)$, the loss function;
- $g(\cdot)$, the gradient;
- $\langle x_i, x_j \rangle$, the dot product of two vectors x_i and x_j.

A. SVM

The loss function and gradient of Support Vector Machine (SVM) over one data point are:

$$l(w, x) = \max(0, 1 - y \cdot \langle w, x \rangle), \tag{3}$$

$$g(w, x) = \begin{cases} -y \cdot x, & \text{if } 1 - y \cdot \langle w, x \rangle > 0, \\ 0, & \text{otherwise.} \end{cases} \tag{4}$$

The statistics to compute is the dot product $\langle w, x \rangle$ of the model w and the feature vector x. The partial statistics is then simply the dot product between the model and data partitions on each worker. The master sums over the dot product and broadcasts the sum back to the workers. Each worker then follows Equation 4 to compute the gradient.

B. LR

The loss function and gradient of Logistic Regression (LR) over one data point are:

$$l(w, x) = log(1 + exp(-y \cdot \langle w, x \rangle)), \tag{5}$$

$$g(w, x) = \frac{-y}{1 + exp(y \cdot \langle w, x \rangle)} \cdot x. \tag{6}$$

Same as SVM, the statistics is the dot product $\langle w, x \rangle$ of the model w and the feature vector x. Each worker computes the partial dot product using its local data and model partitions. The master sums up the partial dot products from the workers. Each worker follows Equation 6 to compute the gradient after receiving the aggregated dot product from the master.

C. MLR

Multinomial Logistic Regression (MLR) is an approach that generalizes logistic regression to work for multiclass classification problems, i.e. with more than two possible discrete outcomes. We assume there are K categories, i.e., $y \in \{1, 2, 3, ..., K\}$. The model parameter of MLR is an $m \times K$ matrix. We use w_k to represent the model parameter for each category $k \in [K]$. The loss function and gradient over one data point are the following:

$$l(w, x) = -\sum_{k=1}^{K} t_k log\left(\frac{exp(\langle w_k, x \rangle)}{\sum_{j=1}^{K} exp(\langle w_j, x \rangle)}\right), \tag{7}$$

$$g_{w_k}(w, x) = \left(\frac{exp(\langle w_k, x \rangle)}{\sum_{j=1}^{K} exp(\langle w_j, x \rangle)} - t_k\right) \cdot x. \tag{8}$$

Here $t_k = 1$ if the label of x is k, otherwise $t_k = 0$.

The statistics remain to be the dot products, and the distributed procedure of computing gradients is similar to that of LR. The only difference is that in MLR, for each data point, there are K (rather than one) statistics from each worker to be sent through the network and aggregated by the master.

D. FM

Factorization Machine (FM) [13], [25] can further capture interactions between features and is widely used in recommendation systems. According to Rendle [13], an FM model of degree $d = 2$ over one data point can be expressed as follows:

$$\hat{y}(x) = \langle w, x \rangle + \sum_{i=1}^{m} \sum_{j=i+1}^{m} < v_i, v_j > \cdot x_i \cdot x_j. \tag{9}$$

Here w captures the importance of each feature and $\langle v_i, v_j \rangle$ captures the importance of the interaction between two features. Training an FM model means to learn the parameters $w \in \mathbb{R}^m$ and $V \in \mathbb{R}^{m \times F}$, where F is the number of factors (a hyper parameter of FM).

To understand what "statistics" should be computed for FMs, we further rewrite the above equation as

$$\hat{y}(x) = \sum_{i=1}^{m} \left(w_i \cdot x_i - \frac{1}{2} \cdot \sum_{f=1}^{F} v_{i,f}^2 \cdot x_i^2\right) + \frac{1}{2} \cdot \sum_{f=1}^{F} \left(\sum_{i=1}^{m} v_{i,f} \cdot x_i\right)^2. \tag{10}$$

Assume that we use logistic loss for FMs. The loss function and gradient over one data point are then:

$$l(w, x) = log(1 + exp(-y \cdot \hat{y}(x)), \tag{11}$$

$$g(w, x) = \frac{-y}{1 + exp(y \cdot \hat{y}(x))} \cdot x, \tag{12}$$

$$g(v_{i,f}, x) = \frac{-y}{1 + exp(y \cdot \hat{y}(x))} \cdot \left(x_i \sum_{j=1}^{m} v_{j,f} x_j - v_{i,f} x_i^2\right). \tag{13}$$

Implementation of FMs in ColumnSGD is a bit more complicated compared with LR and SVM. As shown in Equation 10, for each data point we need to aggregate two statistics from the workers: (1) the dot product over w minus half of the squared dot product over V; and (2) the dot product over V in each latent space. To recover the gradients following Equations 12 and 13, we need to broadcast $F+1$ statistics for each data point, i.e., the $\hat{y}(x)$ and the dot product over V in each latent space in Equation 10.

```
1  /* worker functions */
2  def initModel(val K: Int) = {
3    // initialize the model as an array
4    val local_model = Array.ofDim(K)
5    local_model.initialize()
6  }
7  def computeStat(val batch_data: Array[DataPoint]) = {
8    // compute the partial statistics using
9    // local data and local model
10   val local_stat = Array.ofDim(batch_size)
11   for (id <- 0 until batch_size)
12     local_stat(i) = local_model.dot(batch_data(id))
13   return local_stat
14 }
15 def updateModel(val stat: Array[Double],
16   val batch_data: Array[DataPoint]) = {
17   // compute the gradient and update the model
18   val local_model_dim = num_features / num_workers + 1
19   val grad = Array.ofDim(local_model_dim)
20   for (id <- 0 until batch_size){
21     val y_i = batch_data(id).label
22     grad += -y_i / (1 + exp(y_i) * stat(id))
23       * batch_data(id).features
24   }
25   local_model -= stepsize * grad / batch_size
26 }
27 ======================================================
28 /* master functions */
29 def reduceStat(val stat1: Array[Double],
30   val stat2: Array[Double]) = {
31   // aggregate statistics from workers
32   return (stat1 + stat2)
33 }
```

Fig. 12. Training LR using SGD in ColumnSGD.

IX. Programming Interface

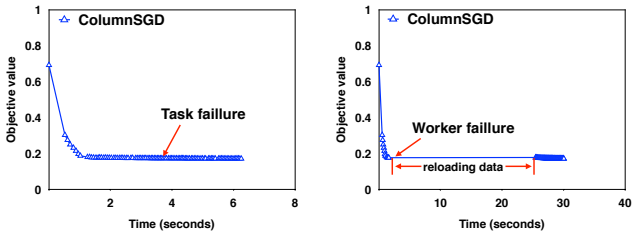
We showcase how to implement LR in ColumnSGD, by walking through the example Scala code in Figure 12.

- `initModel` (lines 2-6): We instantiate the model on each worker as an array through this function. The dimension K is the number of features owned by each worker.
- `computeStat` (lines 7-14): We compute the statistics on each worker through this function. In LR the statistics are in the form of dot products (see Equation 2). For each data point in the mini-batch, we compute a partial dot product for the dimensions corresponding to the local model.
- `reduceStat` (lines 28-33): The master aggregates the statistics from each worker through this function. In LR, the master simply sums up the partial dot products received from the workers and broadcasts the sum to all workers.
- `updateModel` (lines 15-26): After each worker receives the complete dot product from the master, this function first computes the gradients using the data batch and the dot product, and then uses the gradients to update the model.

X. Fault Tolerance

There are three kinds of failures in ColumnSGD.

1) **Task Failure.** All ColumnSGD needs to do is to start a new task when a (Spark) task fails on a worker. No additional work on data loading and partitioning is required, as both the training data and the model are stored on the same worker when the new task starts.
2) **Worker Failure.** If a worker fails, both partitions of the model and the training data on this worker are lost. In ColumnSGD we do not checkpoint the model periodically as other state-of-the-art [4]–[7]. Rather, we rely on the robustness of SGD [22] for fault tolerance. Even if a limited number of updates are incorrectly computed, SGD is still guaranteed to converge to the same optimum though it may take more iterations. Therefore, upon a worker failure, one can just reload the training data and randomly assign some values (e.g., all zeros) to this model partition.
3) **Master Failure.** If the master fails, we have to restart the whole job because we rely on the master for scheduling.



(a) Task failure (b) Worker failure

Fig. 13. Fault tolerance of ColumnSGD.

We examine fault tolerance in ColumnSGD by testing its performance upon task failure and worker failure. To emulate task failure, we simply throw an exception during the training process. To emulate worker failure, we randomly pick one worker and kill it. Figure 13 presents the results when training LR on `kdd12` in **Cluster 1**.

(Task Failure) As shown in Figure 13(a), ColumnSGD is not affected by task failure at all. Since we cache the training data and the model in memory, when a task fails we only need to launch a new task to resume training.

(Worker Failure) ColumnSGD reloads training data upon worker failure. Figure 13(b) shows that ColumnSGD needs about 23 seconds to reload the data shard on that particular failed worker. However, ColumnSGD can converge to the optimal solution even if we do not perform checkpointing.