PS2: Parameter Server on Spark

Zhipeng Zhang[†§] Bin Cui[¶†] Yingxia Shao[‡] Lele Yu[§] Jiawei Jiang[§] Xupeng Miao[†§]

[†]School of EECS & Key Laboratory of High Confidence Software Technologies (MOE), Peking University
[¶]Center for Data Science & National Engineering Laboratory for Big Data Analysis and Applications, Peking University, [‡]Beijing Key Lab of Intelligent Telecommunications Software and Multimedia, Beijing University of Posts and Telecommunications, [§]Tencent Inc.
{zhangzhipeng, bin.cui, xupeng.miao}@pku.edu.cn
shaoyx@bupt.edu.cn, {leleyu,jeremyjiang}@tencent.com

ABSTRACT

Most of the data is extracted and processed by Spark in Tencent Machine Learning Platform. However, seldom of them use Spark MLlib, an official machine learning (ML) library on top of Spark due to its inefficiency. In contrast, systems like parameter servers, XGBoost and TensorFlow are more used, which incur expensive cost of transferring data in and out of Spark ecosystem.

In this paper, we identify the causes of inefficiency in Spark MLlib and solve the problem by building parameter servers on top of Spark. We propose PS2, a parameter server architecture that integrates Spark without hacking the core of Spark. With PS2, we leverage the power of Spark for data processing and ML training, and parameter servers for maintaining ML models. By carefully analyzing Tencent ML workloads, we figure out a widely existing computation pattern for ML models—element-wise operations among multiple high dimensional vectors. Based on this observation, we propose a new data abstraction, called Dimension Co-located Vector (DCV) for efficient model management in PS2. A DCV is a distributed vector that considers locality in parameter servers and enables efficient computation with multiple co-located distributed vectors. For ease-of-use, we also design a wide variety of advanced operators for operating DCVs. Finally, we carefully implement the PS2 system and evaluate it against existing systems on both public and Tencent workloads. Empirical results demonstrate that PS2 can outperform Spark MLlib by up to 55.6× and specialized ML systems like Petuum by up to 3.7×.

SIGMOD '19, June 30-July 5, 2019, Amsterdam, Netherlands

ACM ISBN 978-1-4503-5643-5/19/06…$15.00
https://doi.org/10.1145/3299869.3314038

CCS CONCEPTS

• **Computer systems organization** → *Distributed architectures.*

KEYWORDS

Spark; Parameter Server; Distributed Machine Learning

ACM Reference Format:
Zhipeng Zhang, Bin Cui, Yingxia Shao, Lele Yu, Jiawei Jiang, Xupeng Miao. 2019. PS2: Parameter Server on Spark. In *2019 International Conference on Management of Data (SIGMOD '19), June 30-July 5, 2019, Amsterdam, Netherlands.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3299869.3314038

1 INTRODUCTION

Nowadays using machine learning to train high-dimensional models from large datasets has been a standard task in current industrial applications [15, 20, 25, 26, 28]. In recent years, there is a flurry of work on building specialized systems to manage large-scale ML models, which often employ a parameter server architecture [4, 7, 16, 18, 28]. However, the challenge of distributed ML in action comes from not only the efficiency of training ML models, but also the capability of (pre)processing training data.

(Real Task In Tencent.) *For user profiling, it is common to use different ML models to learn from multiple data sources.* Specifically, we use graph embedding [12, 23, 27] to learn the latent information in large social networks (e.g., the QQ social network contains more than 800 million users) and text mining [29, 30] to learn user interests from users' texts. Furthermore, to employ high-dimensional user profiling for personal recommendation, where each user instance may contain more than 200 million features, classification models like logistic regression or factorization machine are used.

Spark [32] is a fundamental computation framework for big data analytics in Tencent, and naturally we use it to

collect and clean various kinds of aforementioned data like graphs, texts and images, and create the large-scale training data. However, when training ML models, Spark MLlib [22], an official ML library on top of Spark, is inefficient to be used in practice. In contrast, we use specialized ML systems based on parameter servers [21] for efficient ML training. However, this solution with two separated systems not only incurs the expensive cost for data movement between Spark and parameter server-based ML systems, but also increases the complexity of distributed ML pipeline.

In this paper, we aim for designing *a system that can leverage the benefits of both big data processing and efficient ML training*. We first identify that the inefficiency of Spark MLlib comes from the "single-node" bottleneck in the network communication step. When training an ML model using MLlib, a single node (i.e., the driver in Spark) needs to collect updates from all workers and broadcasts model back to all of the workers, whose size is usually linear to the dimension of the features. As a result, when the number of features increases to millions or tens of millions, the communication cost between the driver and dozens or hundreds of workers would dominate the whole training process.

Our solution is to build a parameter server module on top of Spark. Conceptually, we use multiple servers to replace the single-node driver for maintaining the ML model, thus the communication bottleneck can be removed. There remain two challenges for building such a system:

- *Ease-Of-Use.* The new system should be easy to use and fully compatible with the Spark ecosystem. Users can deal with large datasets and train ML models in a single system. Furthermore, the system modification over Spark should be transparent to the Spark users.
- *High Efficiency.* The new system should be efficient for various real-world ML workloads in Tencent. To achieve this, model-specific optimization techniques need to be carefully designed.

Existing solutions like Glint [14] and DistML[1] cannot address these two challenges. For example, Glint hacks the Spark system, and users have to manually start the parameter servers before they submit a job. Moreover, the existing systems only provide the pull/push interfaces to users, without considering the diversity of operations on ML models, leading to an inefficient implementation on different workloads.

To tackle the above two challenges, we propose PS2, an easy-to-use parameter server built on top of Spark with high efficiency. PS2 leverages Spark for efficient data processing and parameter servers for model management.

Ease-Of-Use. We do not break the core of Spark, rather, we treat parameter server as a separate module in PS2 and enable the communication between spark executors and parameter servers. To achieve this, we start a client inside each worker to get the address of parameter servers and use the address to communicate with them. We further add a coordinator in PS2 for scheduling the workers and servers to fulfill an ML job. Thus the job execution is transparent to Spark users but arms Spark with the power of training large models. We also implement basic operators in PS2 like pull/push.

High Efficiency. PS2 resolves the "single-point" bottleneck in Spark by using multiple servers to replace the single-node driver for model management. However, the traditional operators of parameter servers (i.e., pull/push) are not enough for efficient operations on model parameters because it lacks the consideration of characteristics of real-world workloads.

We, with a careful analysis on Tencent ML workloads, identify that the operations on ML model parameters are often complicated and traditional operators like pull/push cannot handle them efficiently. These operations can often be modeled as "element-wise operations on multi-vector ML models" (See Section 3.1), in which we need to perform element-wise operations on two or more distributed model vectors. For example, when running graph embedding models like DeepWalk [23], we need to use the embedding vector of one vertex to update the vectors of its co-occurrence neighbors. With operators like pull/push, we have to pull these two vectors from parameter servers, update them in each worker and push the updated vectors back to the servers. These could cause significant communication overhead between workers and servers.

To handle this case, we propose a new data abstraction, namely **D**imension **C**o-located **V**ector (DCV) to model the ML model parameters. A DCV is a distributed vector on parameter servers that enables efficient element-wise operations with multiple DCVs. Considering that the dimension of DCV can often be hundreds of millions in practice, we propose to partition DCV by column. Further, if the same dimensions of two DCVs are located on different servers, it incurs server-server communication to complete the element-wise operations. To avoid such communication cost, we propose to co-locate their same dimensions on the same server. We extend PS2 with the DCV abstraction and a set of operators for efficient model management.

We finally implement PS2, an easy-to-use and highly efficient system that integrates Spark with parameter servers. PS2 achieves 55.6× faster than Spark MLlib on both public and Tencent workloads. To summarize, our contributions are as follows:

- We identify the inefficiency of ML on Spark and bridge this gap by proposing PS2, a parameter server on top of Spark without breaking the core of Spark. PS2 is easy to use and transparent to Spark users.

[1] https://github.com/intel-machine-learning/DistML

- By further analyzing Tencent ML workloads, we find that the model management can be complicated and the classic "pull/push" operators cannot address the requirement of "element-wise operation on multi-vector ML models". We further propose DCV, a new abstraction that allows more complicated server-side computations on model parameters by considering locality in parameter servers. We extend PS2 with DCV and a set of powerful operators.
- With the DCV abstraction, we implement various ML models by considering the characteristics of each model. Notable examples include Logistic Regression (LR), graph embedding models like DeepWalk [23], Gradient Boosting Decision Tree (GBDT) [15, 17] and Latent Dirichlet Allocation (LDA) [29, 30].
- We evaluate PS2 against systems like Spark MLlib, Petuum, XGboost, Glint [14] and DistML. Experimental results show that PS2 can outperform Spark MLlib by up to 55.6× and specialized ML systems by up to 3.7× on both public and Tencent workloads.

PS2 now has been deployed in Tencent for months, tackling real-world large-scale ML tasks like graph embedding, text mining and classification.

Paper Organization. We identify the causes of inefficiency of Spark MLlib using an empirical solution in Section 2. We propose the system design of PS2 in Section 3, followed by the DCV abstraction, with which we better manage ML model parameters in Section 4. We present the system implementation in Section 5 and evaluate PS2 in Section 6. We present related work in Section 7 and conclude in Section 8.

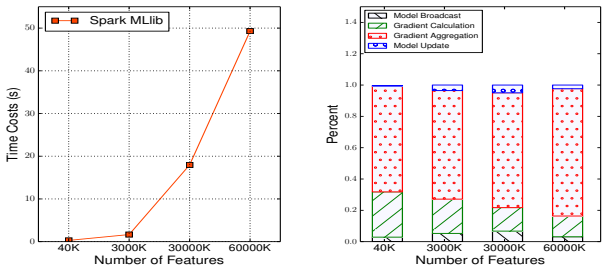
2 ANALYSIS OF SPARK MLLIB

We empirically study the characteristics of MLlib, an official ML library on top of Spark. Specifically, we introduce the process of training ML models in MLlib and use real-world workloads to demonstrate its bottlenecks in ML training.

Execution Process in Spark MLlib. In Spark MLlib, there is a *driver* and multiple *executors*. The driver controls the logic of the computation while the executors run parallel tasks defined in RDD. An iteration of running SGD to train LR in Spark MLlib can be divided into four steps:

(1) *Model broadcast*: The driver first broadcasts the model parameters to all executors;
(2) *Gradient calculation*: Each executor samples a fraction of data points and calculates the gradient;
(3) *Gradient aggregation*: The driver aggregates the gradients from all executors;
(4) *Model update*: The driver uses the aggregated gradients to update the model locally.

Profiling Results of Spark MLlib. In Tencent, logistic regression is widely used for recommendation systems, where



(a) Time Cost per Iteration (b) Details of Time Cost

Figure 1: Empirical analysis for Spark MLlib.

the number of features often gets hundreds of millions. To understand the performance bottlenecks in MLlib, we train LR using SGD on four datasets with various number of features, i.e., $40K$, $3,000K$, $30,000K$ and $60,000K$. We log the time cost for the four steps mentioned above in each iteration and the results are shown in Figure 1. We use 20 executors for each workload and the mini batch fraction is set to 0.01.

Figure 1(a) presents the time cost per iteration, and Figure 1(b) reports the time breakdown of four steps. From Figure 1(a), we observe that the performance of Spark MLlib degrades with the increasing of number of features. For example, the performance is 168× slower when increasing the number of features from $40K$ to $60,000K$. This is due to its unreasonable solution for the gradient aggregation step, which employs a single node, i.e., the driver, to collect gradients. In more details, from Figure 1(b), it is easy to figure out that gradient aggregation becomes the bottleneck—it occupies most time of one iteration. Clearly, Spark MLlib performs bad at the gradient aggregation step.

Summary. The inefficiency of training LR in Spark MLlib comes from the communication step, in which the single-node driver collects gradients from all workers. Thus a probable solution of solving the "single-node" bottleneck is to integrate Spark with parameter servers, the state-of-the-art solution that uses multiple servers to replace the single node.

3 SYSTEM DESIGN

We address the inefficiency of Spark MLlib by building PS2, a system that integrates Spark with parameter servers. We first identify the necessity of enabling *server-side* computation by analyzing the computation pattern on ML model update, which is not yet carefully considered by existing ML systems. Based on this observation, we then present the system architecture of PS2. Finally, we use LR as an example to showcase the execution flow of PS2.

3.1 Computation Pattern of Model Update

We introduce the common computation patterns of updating ML models using LR and graph embedding as two examples.

Example 1: Adam for LR. Logistic Regression is a widely used classification model. Given training data X, the goal of LR is to learn a model parameter w that minimizes the logistic loss over X. Adam [19], a variant of stochastic gradient descent (SGD), is mostly used for optimization due to its fast convergence. Apart from the weight vector, Adam stores two separate vectors to adapt the learning rate on each dimension, i.e., an exponentially decaying average of past squared gradients s_t, and an exponentially decaying average of past gradients v_t. The model w is updated as follows:

$$\begin{aligned}
\vec{s_t} &\leftarrow \beta_1 \vec{s_{t-1}} + (1-\beta_1)\vec{g_t}^2 \\
\vec{v_t} &\leftarrow \beta_2 \vec{v_{t-1}} + (1-\beta_2)\vec{g_t} \\
\vec{s'_t} &\leftarrow \frac{\vec{s_t}}{1-\beta_1^t} \\
\vec{v'_t} &\leftarrow \frac{\vec{v_t}}{1-\beta_2^t} \\
\vec{w_{t+1}} &\leftarrow \vec{w_t} - \frac{\eta}{\sqrt{\vec{s'_t}+\epsilon}}\vec{v'_t}
\end{aligned} \tag{1}$$

where g_t is the gradient of the current iteration, $\beta_1, \beta_2, \epsilon$ are hyperparameters of Adam and η is the learning rate.

The model vectors (i.e., $\vec{s_t}, \vec{v_t}$ and $\vec{w_t}$) share the same dimensions, which is the dimension of features and could be up to hundreds of millions in practice. When updating model $\vec{w}$, we first use the gradient to compute $\vec{s_t}$ and $\vec{v_t}$. We further revise the value of $\vec{s_t}$ and $\vec{v_t}$ and correct the bias. Finally, the weight vector w is updated by the revised vectors. To execute the above equations, element-wise operations among these model vectors are needed.

Example 2: SGD for Graph Embedding. Graph embedding [12, 23, 27] introduces ML for analyzing graphs. Given a graph (V, E) with vertex set V and edge set E, the goal of graph embedding is to learn an embedding vector for each vertex so that the vector can preserve some property about the vertex in this graph. SGD is often employed for training graph embedding. A typical computation pattern is as follows: For each vertex u, we allocate two K-dimensional vectors, one vector $\vec{u}$ as its embedding vector and the other $\vec{v}'$ as its "context" vector, where K is a hyperparameter. During the training process, we sample a pair of vertices (u, v) according to a model-specific rule (e.g., random walk for DeepWalk [23]), and consider this two vertex to be similar. To embed this relationship into their vector representations, we adopt the following update rule:

$$\begin{aligned}
\vec{u_{t+1}} &\leftarrow \vec{u_t} - \eta \cdot (\sigma(\langle \vec{u_t}, \vec{v_t}' \rangle) - 1) \cdot \vec{v_t}' \\
\vec{v_{t+1}}' &\leftarrow \vec{v_t}' - \eta \cdot (\sigma(\langle \vec{u_t}, \vec{v_t}' \rangle) - 1) \cdot \vec{u_t}
\end{aligned} \tag{2}$$

Here $\sigma(x) = 1/(1 + exp(-x))$ is the sigmoid function and η is the learning rate. $\langle \vec{u}, \vec{v}' \rangle$ is the dot product between two vectors. To execute the above equation, we first compute the dot product and use the result to conduct element-wise operations on the two embedding vectors.

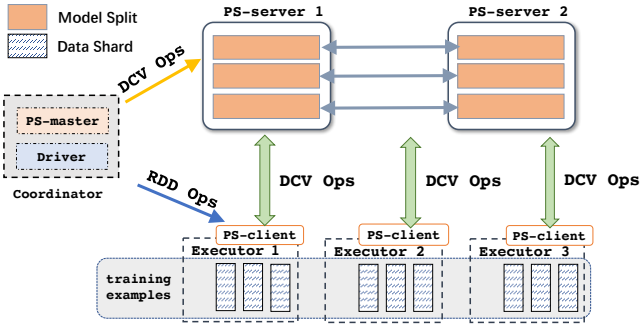


Figure 2: System architecture of PS2.

Summary. We summarize two inherent properties for these operations on ML models, which are not explored in existing ML systems.

- *Multiple Vectors as the Model.* First, these ML models contain multiple vectors as the model. For example, Adam contains one weight vector and two auxiliary vectors (i.e., $\vec{s_t}$ and $\vec{v_t}$), and graph embedding has $2V$ vectors as the model.
- *Element-wise Operations on Model Vectors.* Second, we need element-wise operations on these vectors when performing the model update. For example, in Adam we need to perform element-wise *addition*, *multiplication* and *division* on the four model vectors. For graph embedding, element-wise *addition* and *multiplication* among the embedding vectors are also needed.

These two properties of ML model operations require careful consideration for system design. Maintaining these vectors proposes a heavy storage overhead while the linear algebra operations among these vectors introduce severe computation overhead. Note that, these two properties do not only exist in Adam for LR and SGD for graph embedding. Some other optimization algorithms and machine learning models, such as L-BFGS [8], Adagrad [10], GBDT and LDA also share one or both properties.

3.2 Architecture

To tackle the huge communication overhead in Spark MLlib as well as the storage overhead of model vectors, we propose PS2, a parameter server architecture built on top of Spark. To further address the requirement of element-wise operations among model vectors, we propose a new abstraction, namely Dimension Co-located Vector (See Section 4 for details), to enable server-side computation that considers locality in model management.

Figure 2 presents the system architecture of PS2. There are three main modules, including a coordinator, several workers and servers:

- **Workers:** PS2 employs executors in Spark as workers. Workers are responsible for processing the training data, and calculating the model updates. When pulling models or pushing updates, workers communicate with servers via DCV Ops.
- **Servers:** PS2 relies on parameter servers for model management, including both the storage and computation. The model vectors are maintained as DCVs and distributed among multiple servers. The servers provide access of model vectors to workers or the coordinator through the DCV Ops.
- **Coordinator:** PS2 further adopts a coordinator to schedule the servers and workers.

3.3 Execution Flow

We use training logistic regression with Adam as an example to demonstrate the execution process of PS2. During the initialization phase, we use Spark to load the training data as RDDs [31] (lines 1-2) and initialize model parameters distribtuedly on parameter servers through the DCV abstraction (lines 3-7). Then we iteratively use the training data to correct the model parameters. Each iterations can be divided as the following four steps:

(1) *Model pull* (lines 13-14): Each worker pulls the weight from multiple parameter servers using a DCV *pull* operator.[2]
(2) *Gradient calculation* (lines 15-16): Each worker uses the model pulled and the local batch of training data to compute the gradient.
(3) *Gradient push* (lines 17-18): Each worker pushes its local gradients to parameter servers via a DCV *add* operator. A global barrier (line 19) is further incurred relying on Spark's scheduling mechanism, to ensure that all the gradients are added to the DCV on servers.
(4) *Model update* (lines 21-26): Servers incur an element-wise operation among four DCVs via a *zip* operator to do server-side computation among multiple servers.

4 DIMENSION CO-LOCATED VECTOR

In this section, we introduce the new abstraction called Dimension Co-located Vector, which aims to accelerate the operations on ML models and simplify the programming of server-side computation. We introduce the design considerations for DCV and then propose a set of DCV operators to better support the server-side computation. The corresponding implementation is also included.

[2]Example usage of DCV operators is covered in Section 5.2.

```
1  // load data as RDDs
2  val data = sparkContext.textFile("hdfs://...")
3  // Allocate four DCVs to model vectors
4  val weight  = DCV.dense(dim, 4)
5  val velocity = DCV.derive(weight).fill(0.0)
6  val square  = DCV.derive(weight).fill(0.0)
7  val gradient = DCV.derive(weight)
8
9  for (i <- 0 until numIterations) {
10   gradient.zero()
11   // Gradient computation
12   data.sample(fraction).mapPartition{ case iterator =>
13     // Pull weight from PS
14     val local_weight = weight.pull()
15     // Calculate gradient locally
16     val local_gradient = calculateGradient(local_weight, iterator)
17     // Push local gradient to PS
18     gradient.add(local_gradient)
19   }.foreach()
20
21   // Model update
22   weight.zip(velocity, square, gradient).mapPartition {
23     case (w, v, s, g) =>
24       // Update model in each partition (server-side computation)
25       updateModel(w, v, s, g)
26     }
27 }
```

Figure 3: Code sample of "Adam for LR" on PS2. The underlined methods are DCV operators.

4.1 Core Abstraction: DCV

By analyzing the operations on ML models in Section 3.1, we identify that simple pull/push operators that access models by rows, are not enough for managing ML models in parameter servers. ML models can often be modeled as a combination of multiple vectors and element-wise operations on these model vectors are needed. This actually imposes a requirement for accessing the distributed model parameters by column.

To meet this requirement, we propose a new abstraction, named **D**imension **C**o-located **V**ector (DCV). DCV is a vector distributed among parameter servers, and supports two types of operators. One is row access operator, which provides read/write interfaces to handle data communication between workers and servers. The other is column access operator, which operates the same columns from multiple vectors each time. Furthermore, to support efficient element-wise operations on multiple vectors, we partition one DCV with a column-partition strategy and co-locate the same dimensions of multiple DCVs onto the same server.

4.2 Operator Sets

For ease-of-use, PS2 provides a set of operators over DCV. As mentioned before, the operators are divided into two types according to their access patterns. Table 1 lists a set of operators in PS2.

Category	Operators
Row Accessing Ops	pull, push, sum, nnz, norm2
Column Accessing Ops	axpy, dot, copy, sub, add, mul, div
Creation Ops	derive, dense, sparse

Table 1: DCV Operators

Row Access Operators. These operators provide read-/write operations for one row and handle data communication through network. There are mainly two scenarios that require row access operators. The first one is to transfer models between workers and servers, including *pull* and *push*. The second is to obtain an aggregation value from one vector, such as *sum* and *norm2*. Note that existing parameter servers also support row access operators.

Column Access Operators. These operators operate the same columns of multiple DCVs. They can perform element-wise operations, such as *dot* and *add*, among multiple vectors. These operators can be used to better support server-side computation for ML models.

Apart from row/column access operators, we also provide some operators to create DCVs. PS2 provides normal operators like *sparse* and *dense* to create sparse and dense model vectors. More than that, PS2 supports a special operator, called *derive*. With the *derive* operator, users can create a DCV that is co-located with a given DCV. This enables us to explore efficient element-wise operations for two DCVs without heavy communication cost accross servers.

4.3 Implementation of DCV

The performance of algorithms in PS2 is heavily influenced by the performance of row access operators and column access operators of DCVs. To achieve efficient row access operators, we use column-partition strategy, to divide one vector into multiple splits and store the splits among servers. Thus, we can execute row access operators (e.g., pull/push) in parallel across workers and servers. And to speed up column access operators, we co-locate the same dimensions of multiple vectors on the same server. Thus, PS2 avoids communication among servers when executing element-wise operations on multiple DCVs (e.g., add/mul). This requirement leads us to a new operator for DCVs, namely *derive*.

Column-partition Strategy. As mentioned in Section 4.1, an ML model often contains a set of vectors. The existing parameter server systems, such as Petuum, usually partition the model matrix by row. That is, each partition is actual a complete vector. Thus, the system cannot run row access operators in parallel, causing single-point problem.

Instead, DCV in PS2 uses a column partition strategy [3]. With such partition strategy, we not only accelerate the row access operators by storing one row with multiple servers,

```
1  @Inefficient writing
2  val v1 = DCV.dense(dimension)
3  val v2 = DCV.dense(dimension)
4  val dot1 = v1.dot(v2)
5
6  @Correct writing
7  val v1 = DCV.dense(dimension)
8  // derive a new DCV from an existing one
9  val v2 = DCV.derive(v1)
10 val dot2 = v1.dot(v2)
```

Figure 4: "dot" between two DCVs. Lines 1-4 show an inefficient usage of DCV while lines 6-10 present the correct way.

but also open up the opportunity for optimizing column access operators. Through column partitioning, DCVs become a set of vector splits. If we cleverly assign these splits, i.e., co-locating splits with the same dimensions onto the same server, then we can avoid the communication cost when using column access operators to operate on multiple DCVs. The co-located property is achieved by the *derive* operator.

Derive Operator. The *derive* operator is to generate co-located DCVs. Specifically, when allocating one DCV through *dense* operator, we create a distributed raw model matrix with k rows, in which $(k-1)$ rows are pre-allocated for future usage. Thus, when calling the *derive* method, one free row from the matrix is returned, and the new derived DCV is guaranteed to share the same partition strategy with the first row in the raw matrix. The initial size of the matrix (i.e., the k) is usually small, for example ten. Programmers can also pass another parameter to precisely set the number of rows in the original matrix.

We take a simple binary operator *dot* to demonstrate the power of DCVs using Figure 4. Lines 1-4 show an inefficient programming of performing *dot* between two DCVs. The two DCVs are irrelevant because they do not share the same partition strategy. It is possible that same dimensions of v_1 and v_2 locate on different servers, and would incur huge communication cost among parameter servers when performing *dot*. Lines 6-10 give the right way, which uses the *derive* operator. In line 9, v_2 is generated by calling the *derive* method of DCV with v_1 as the parameter. Through this way, their partition strategies are guaranteed to be the same. Thus, the *dot* between v_1 and v_2 is efficient and there is no data shuffling across servers during the execution.

5 SYSTEM IMPLEMENTATION

PS2, as an industrial system that brings Spark the power of training large scale ML models with high efficiency, has a careful engineering design for system architecture, algorithm optimization and the ability to handle failures.

5.1 Architecture Implementation

We implement PS2 by building a parameter server module on top of Spark. In general, Spark is used for processing

and holding immutable training data while parameter server is responsible for maintaining mutable and shared model parameters. To simplify the programming of PS2 and make it easy to use for Spark users, we implement the architecture by launching Spark and parameter server as two separated applications. To enable the communication between workers and servers, we build a PS-client on each worker. Further, for the simplicity of system architecture, we use the driver to complete the functions of the coordinator, which means that the driver needs to manage and monitor the running of servers. Therefore, PS2 has four components, i.e., *PS-server*, *PS-client*, *Executors* and the *Coordinator* in Figure 2.

- **Executors.** PS2 employs executors in Spark as workers. They leverage the power of RDD [31] to handle large scale data processing and computation of model updates.
- **PS-servers**. PS-servers store model parameters. During the training process, the model parameters are partitioned into multiple shards and each PS-Server maintains a fraction of them.
- **PS-client**. PS-client is the bridge module between PS-server and executors. Each executor contains a PS-client and is used to issue DCV Ops to PS-servers, to operate on the model parameters.
- **Coordinator**. The coordinator is extended from the driver in Spark. It schedules workers and servers to fulfill an ML job, such as issuing RDD operators to workers, DCV operators to servers and synchronizing among servers and workers. Apart from scheduling executors, it also contains a new module called *PS-master*. *PS-master* manages the lifetime of *PS-servers*, and provides some meta information, including the locations and routing tables for *PS-client* to locate parameters.

We implement PS2 by Java and Scala. The RPC framework in PS2 is implemented by Netty[3] and Protobuf[4], while the data communication between parameter servers and Spark is directly transferred through Netty. The modification of PS2 over Spark is transparent to Spark users and programmers can submit PS2 jobs similarly as they do for Spark. Moreover, since parameter server is launched as a separated application, PS2 is compatible with any version of Spark.

5.2 Algorithm Implementation

In this section, we present the algorithm implementations of real-world workloads in Tencent. For each workload, we introduce the ML model parameters and present how to implement them with DCVs in PS2.

[3]https://netty.io/

[4]https://developers.google.com/protocol-buffers/

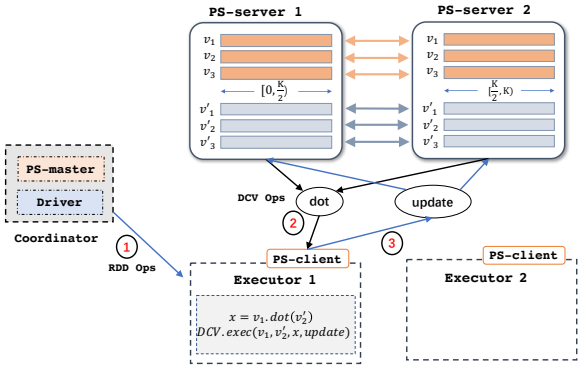


Figure 5: Implmentation of DeepWalk.

5.2.1 Adam for LR. The model parameters for training LR using Adam are four vectors (See Section 3.1). Moreover, element-wise operations across these model vectors are needed. In a industrial workload, the number of features can often be hundreds of millions or bigger.

In PS2, we model the four model vectors as DCVs to make sure that they are distributedly co-located on the servers. Figure 3 shows the implementation of using Adam to train LR. We first allocate one DCV to store the weight vector and use *derive* to generate the other three co-located vectors (lines 3-7). Next, each worker pulls the weight vector distributedly from the servers and samples a mini batch of training data to calculate the gradients (lines 12-16). After that, each worker pushes the gradient to servers via the DCV operator *add* (line 18). Note that we incur a barrier here using Spark's *foreach* operator to ensure that all gradients are pushed to the servers. Finally, we incur a *zip* operator across the four vectors and use gradient to update the other three vectors (lines 21-26). Due to the ability of server-side computation enabled by DCVs, no extra communication overhead across servers is needed.

5.2.2 DeepWalk. Given a graph with V vertices, the model parameters for graph embedding are $V \times 2$ vectors, each of which is a K dimensional vector. Here K is a hyper parameter, which could be one hundred or bigger. During the training process, we need to conduct element-wise operations on model vectors. A normal approach is to put the $2V$ model vectors in parameter servers. When updating the embedding vectors of two vertices, we pull them from the parameter servers, update them locally and push them back to the servers. However, this could lead to huge communication overhead when K is big.

Figure 5 shows the process of running DeepWalk in PS2. In PS2, we model the $2V$ embedding vectors as $2V$ DCVs. For example, these vectors are partitioned by column and placed on two servers. After the coordinator issues one executor to update the model vectors of two vertices, the executor incurs a DCV *dot* operator to compute the dot product of these two

```
// Allocate DCVs to store the embedding vectors for nodes
val first = DCV.dense(K, V*2)
val embeddings = new Array[DCV](V*2)
embeddings(0) = first
for (i <- 1 until V*2) embeddings(i) = DCV.duplicate(u)

// processing the input data to sample the similar node pairs
val data = calculateSimilar(sc.textFile(input))

for (i <- 0 until numIterations) {
  data.map { case (u, v) =>
    val input_u = embeddings(u)
    val output_v = embeddings(v + V)
    // dot product
    val dot = input_u.dot(output_v)
    val sig = 1 - sigmoid(dot)
    // update
    input_u.iaxpy(output_v, sig*eta)
    output_v.iaxpy(input_u, sig*eta)
    // loss value
    calculateLoss(dot)
  }.sum()
}
```

Figure 6: Code sample for the Graph Embedding algorithm on PS2. Function *calculateSimilar* is employed to generate similar pairs from a graph. Function *calculateLoss* is used to calculate the loss value for current pair.

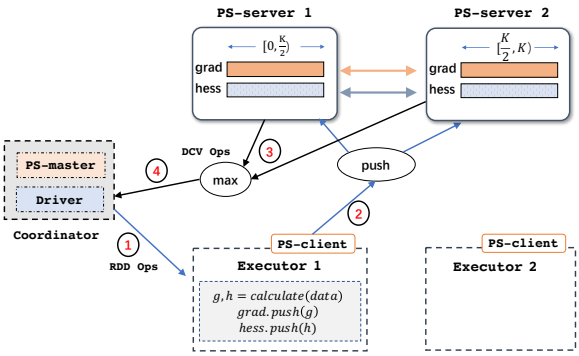


Figure 7: Implmentation of GBDT. "grad" and "hess" are first-order and second-order gradient histograms, respectively.

embedding vectors, via server-side computation. After that, the executor triggers another DCV operation to update these two embedding vectors. Here we use a self-defined function to specify the updating rule according to Equation 2. With the power of DCV, we reduce the communication overhead of pulling/pushing the embedding vectors, rather, only some scalars are transferred through network.

5.2.3 GBDT. For the training of GBDT, two core operations are histogram construction and split finding. In histogram construction, GBDT builds multiple vectors, each of which is a data structure called gradient histogram. Afterwards, a split criterion, including one feature and its corresponding value, is calculated with these vectors. Similar as Adam and DeepWalk, this multi-vector case requires element-wise operations between vectors.

```
// Allocate two DCVs to store gradient histograms
val gradHist = DCV.dense(dim, 2).fill(0.0)
val hessHist = DCV.derive(gradHist).fill(0.0)

for (i <- 0 until numTreeNodes) {
  gradHist.zero()
  hessHist.zero()
  data.mapPartition { case iterator =>
    // Build histograms
    val localGrad = buildGrad(iterator)
    val localHess = buildHess(iterator)
    gradHist.add(localGrad)
    hessHist.add(localHess)
  }.foreach()

// Find the optimal split criterion
val maxGain = gradHist.zip(hessHist).mapPartition {
case (grad, hess) =>
    computeInfoGain(grad, hess)
  }.max()
}
```

Figure 8: Code sample for the GBDT algorithm on PS2. Function *buildGrad* and *buildHess* are used to calculate gradient and build histograms.

Figure 7 demonstrates the implementation of GBDT in PS2. We use the parameter server to accomplish the histogram construction and split finding. In order to enable the computation between histogram vectors, we use DCVs to store these histograms, i.e., first-order and second-order gradient histogram. First, we start the task on each worker via a RDD operator. Then, we parallel the gradient calculation among all data partitions and *push* the local gradient to servers. Third, we use the interface of DCVs to complete the split finding (the *max* operator[5]). Due to the server-side computation ability enabled by DCVs in PS2, we can conduct the split finding operation for one feature on servers, avoiding the transmission of gradient histograms.

5.2.4 Other Models. Apart from the above models, we also implement other ML models like LDA, Support Vector Machine, etc. Some modern optimizations are also implemented, like Adagrad, RMSProp and L-BFGS.

5.3 Fault Tolerance

As an industrial system, PS2 also provides mechanisms to handle fault tolerance. There are four types of failures:

Task Failure. Tasks in executors may encounter failures due to unpredictable reasons. During the training process, each task samples a mini batch of training data and computes the model update using the model pulled from PS-servers. After

[5]In this operator, we enumerate the same elements of *grad* and *hess* from left and right, calculate a loss gain, and find the place that yields the maximal loss gain.

this, the model update is pushed to PS-servers via DCV Ops. To handle this failure, PS2 relies on Spark's mechanism to restart a new task. This is correct because the gradient will not be pushed to servers twice—the push operator is the last operation for a task.

Executor Failure. When an executor fails, the training data maintained by the executor are lost. To handle this case, PS2 relies on the fault tolerance provided by RDDs. It simply launches a new executor and reload that partition of training data from the input (e.g., HDFS).

Server Failure. The failure of a server can be detected by the coordinator. When a server is down, it loses its states—a fraction of model parameters. To handle this failure, PS2 periodically checkpoints the model parameters on each server to a reliable external storage. When a server failure happens, the coordinator starts a new server and the new server recovers the latest model by loading from the checkpoints.

Coordinator Failure. For failures that happen to the coordinator, we cannot cope with them. Fortunately, the coordinator handles little workloads and the probability of encountering a failure is low.

6 EMPIRICAL EVALUATION

In this section, we evaluate PS2 from three aspects. First, we demonstrate the benefits brought by DCV. Then we show the efficiency and generality of PS2 by comparing with other existing ML systems. Finally we evaluate the scalability and fault tolerance of PS2.

6.1 Experimental Setup

Experimental Environment. We conduct the comparison on an inner-shared Yarn cluster in Tencent, which contains 2700 machines and executes thousands of jobs every day. Each machine has a 2.2GHz CPU with 12 cores, 256GB memory and 12 × 2TB SATA hard disk. These machines are connected with 10Gbps Ethernet network.

Datasets and ML Models. Table 2 summarizes the datasets used in the experiments. We use three public datasets, such as KDDB, KDD12 and PubMED, as well as five datasets from Tencent, including CTR, APP, Gender, Graph1 and Graph2.[6] For ML models, we evaluate LR, DeepWalk, GBDT and LDA.

Baseline Systems & Evaluation Metrics. We compare PS2 with Spark MLlib 2.1.1, DistML, Glint [14], Petuum 1.1 and XGboost 0.7. DistML and Glint are two pioneering systems that combine parameter server and Spark without server-side computation ability. They rely on Spark 1.6.0 and cannot run over the latest version of Spark. The experimental results of PS2 are conducted based on Spark 2.1.1.

[6]We do not have the original graph. The users from business unit do the sampling of random walks on graphs.

Model	Dataset	#rows	#cols	#nnz	Size
LR	KDDB	19M	29M	585M	4.8GB
	KDD12	149M	54.6M	1.64B	21GB
	CTR	343M	1.7B	57B	662.4GB
LDA	PubMED	8.2M	141K	737M	4GB
	App	2.3B	558K	161B	797GB
GBDT	Gender	122M	330K	12.17B	145GB
Model	**Dataset**	**#vertices**	**#walks**	**Size**	
DeepWalk	Graph1	254K	308K	100MB	
	Graph2	115M	156M	10.5GB	

Table 2: Dataset Statistics. *nnz* denotes number of non-zero items and #walk represents the number of random walks sampled for each graph.

System	LR	DeepWalk	GBDT	LDA
Spark MLlib	✓	✗	✓	✓
DistML	✓	✗	✗	✓
Glint	✗	✗	✗	✓
Petuum	✓	✗	✗	✓
XGboost	✗	✗	✓	✗
PS2	✓	✓	✓	✓

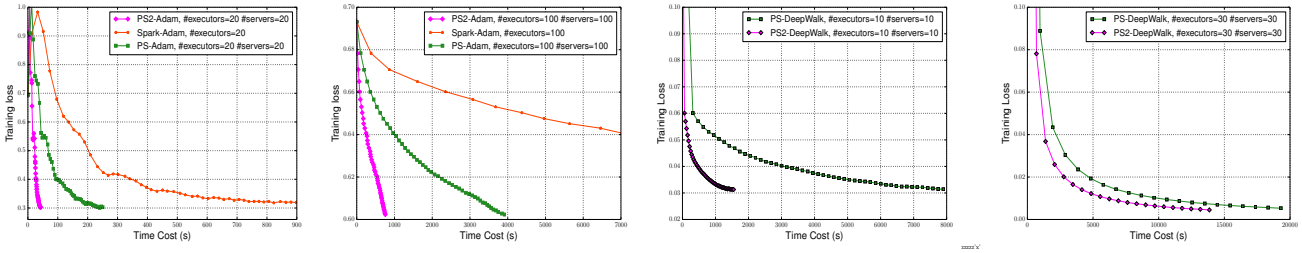
Table 3: Algorithms supported by different systems.

We compare each model mentioned above if the system has implemented it. Table 3 lists the models supported by these systems. To measure the end-to-end performance, we report the loss of each workload as time elapses.

Parameter Settings. When evaluating different systems, we allocate same number of workers/servers and enough memory to guarantee good performance. We follow previous studies [15, 18, 23, 29] to set ML-related hyperparameters for fair comparison. For each workload, we set the same hyper parameters for all systems because these systems enjoy the same statistical efficiency. Details about hyperparameter settings for each workload can be found in Appendix A.

6.2 Evaluation of DCV

We showcase the benefits of the DCV abstraction using LR and DeepWalk. There are two benefits of DCV. First, it resolves the "single-point" bottleneck in Spark by using multiple servers to replace the single-node driver. Second, it enables server-side computation, which could further reduce the communication overhead between the workers and servers. Thus we compare three different realizations of LR and DeepWalk on PS2: (1) purely rely on Spark (namely "Spark-"), (2) using PS2 with pull/push (namely "PS-") and (3) using PS2 with the DCV implementation (namely "PS2-"). For example, we refer to the Adam implementation using PS2 with pull/push as PS-Adam.



(a) Adam (KDDB) (b) Adam (CTR) (c) DeepWalk (Graph1) (d) DeepWalk (Graph2)

Figure 9: Effectiveness of DCV.

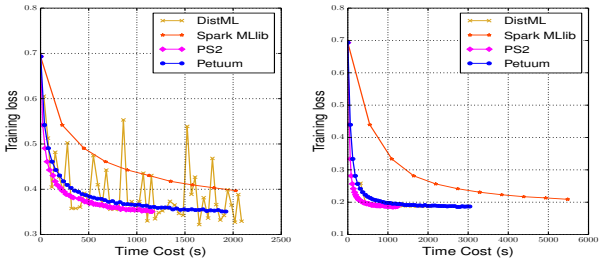
6.2.1 Adam for LR. Figure 9(a) and 9(b) present the effects of DCV when training LR using Adam on KDDB and CTR dataset. The result on KDD12 dataset is similar.

Figure 9(a) shows the evaluation result over KDDB dataset. As we can see, in order to achieve 0.3 training loss, PS2-Adam requires 59 seconds while PS-Adam requires 277 seconds. For Spark-Adam, it requires 926 seconds. We analyze the results in two aspects. First, PS2-Adam is significantly faster than Spark-Adam, by 15.7×. This is because Spark-Adam relies on the driver to broadcast the weight and collect the gradients. When the size of the model is big, it would cause severe overhead on storage and communication on the driver, which is the "single-point" problem as we discussed in Section 2. Second, PS2-Adam is faster than PS-Adam by 4.7×. This is because PS-Adam lacks the ability to do *server-side* computation. It has to pull the gradient as well as the model onto each worker, update the model and push the model back to the parameter server. This incurs significant overhead on communication. However, PS2 utilizes the DCV abstraction to enable *server-side* computation and thus reduces the communication overhead.

Similar facts can be observed on CTR dataset. As shown in Figure 9(b), PS2-Adam is 5× faster than PS-Adam and 55.6× faster than Spark-Adam. This is more impressive and reasonable because the model size is much bigger and Spark-Adam suffers from the "single-point" problem.

6.2.2 DeepWalk. We now evaluate the effects of DCV on graph embedding using DeepWalk [23] as an example. The result of Spark MLlib is not presented because it does not support graph embedding. Figure 9(c) and 9(d) compare PS2-DeepWalk with PS-DeepWalk on two graphs.

In Figure 9(c), PS2-DeepWalk gets 5× faster than PS-DeepWalk on Graph1. The speedup comes from the DCV's ability of conducting server-side computation. As we discussed in Section 5.2, we can avoid transferring the embedding vectors through the network with the DCV abstraction, by conducting server-side "dot" operator. However, for PS-DeepWalk we have to pull the model vectors from the servers, update them in each worker and push the updated vectors back to the servers.



(a) LR (KDDB) (b) LR (KDD12)

Figure 10: End-to-end performance comparison among PS2, DistML, Spark MLlib and Petuum over LR. The number of executors/servers are 20.

In Figure 9(d), PS2-DeepWalk also outperforms PS-DeepWalk, but the speedup is only 1.4×. The reason is that on Graph2, we use 30 servers, which leads the speedup being marginal. Specifically, the benefit of DCV can be degraded by the large number of servers in PS2-DeepWalk. When computing the dot product of two DCVs, we have to collect partial of the dot products from all servers. Thus the communication cost increases when more servers are included. In contrast, for PS-DeepWalk, using more servers leads to more balanced communication when pulling/pushing the model. The trade-off between the PS2-DeepWalk and PS-DeepWalk remains elusive, and we leave this as an interesting future work.

6.3 End-to-End Comparison

We demonstrate the superiority of PS2 by presenting the end-to-end comparison between PS2 and other ML systems.

6.3.1 Comparison On LR. Figure 10 compares the performance of PS2, Spark MLlib, DistML and Petuum on KDDB and KDD12 dataset on training LR using SGD. Adam is not adopted because most of the these systems do not support Adam. The result on CTR dataset is not presented since Petuum cannot be deployed in an industrial environment and DistML always fails to run on CTR dataset with some bugs we cannot fix.

We have the following observations. First, PS2 converges the fastest among the four systems. For example, PS2 gets

speedup by 1.6× and 2.3× over Petuum on KDDB and KDD12 dataset, respectively. The speedup mostly comes from the careful engineering effort. When pulling model vectors from parameter server, PS2 supports sparse communication and only pulls the needed model parameters. However, Petuum has to pull all of the model.

Second, Spark MLlib converges the slowest. As we discussed in Section 2, the bottleneck of Spark MLlib comes from the communication step on the single-node driver. Thus the time cost per iteration is much more expensive than parameter server-based solutions. Furthermore, DistML is not robust. For example, the result of DistML on KDDB dataset in Figure 10(a) cannot converge although we carefully tune the hyperparameters.

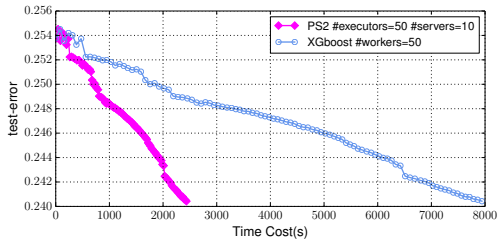


Figure 11: Evaluation over GBDT. The result of Spark MLlib is not presented since it runs out of memory on Gender dataset.

6.3.2 Comparison On GBDT. Figure 11 shows the evaluation over GBDT on Gender dataset. We can see that the performance of PS2 is 3.3× faster than XGboost. To build 100 trees, PS2 requires 2435 seconds while XGBoost costs 7942 seconds. The main contribution of this improvement is caused by the deployment of parameter server architecture. One bottleneck of GBDT is to find out the best split point for each tree node. In XGboost, this phase is conducted by AllReduce, which generates vast communication cost and causes the performance degradation. In PS2, the deployment of parameter server and the DCV abstraction reduce the communication cost and thus achieves speedup. We do not give the comparison between PS2 and Spark MLlib since Spark MLlib always fails due to the Out-of-Memory exception on this dataset.

6.3.3 Comparison On LDA. We proceed to present the evaluation results over LDA. We conduct these experiments using two datasets, PubMED and App. For PubMED dataset, we compare PS2 with Petuum, Glint and Spark MLlib. We set the number of topics as 1000 for comparing Petuum and Glint and 100 for Spark MLlib because Spark MLlib cannot deal with large models. The result of DistML is not presented because it always fails. For App dataset, we only report the performance of PS2 to demonstrate its superiority since other systems cannot handle it.

Figure 12(a) compares PS2 with Petuum and Glint. We can see that the time cost of convergence are 386 seconds, 1440 seconds and 3500 seconds for PS2, Petuum and Glint, respectively. That is, PS2 is 3.7× faster than Petuum and 9× faster than Glint. Although they all employ the parameter server architecture to distribute the communication, PS2 has a more careful engineering effort for its sparse communication implementation and message compression technique.

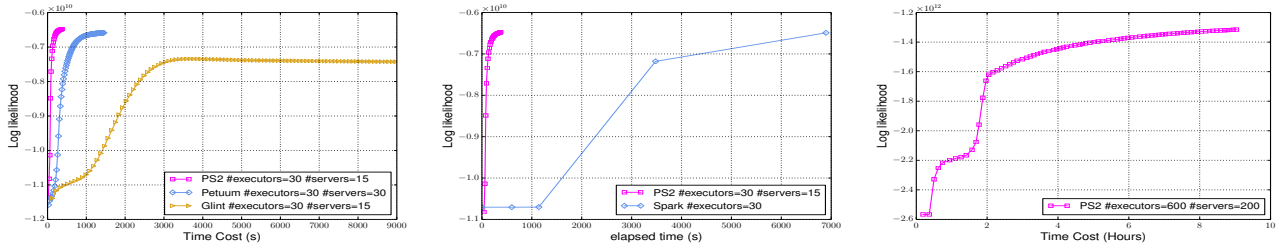
Figure 12(b) shows the comparison between PS2 and Spark MLlib. Spark MLlib requires 6894 seconds to converge and PS2 is 17× faster than MLlib. This is due to the heavy communication overhead on MLlib. PS2 uses multiple servers to replace the single-node driver for managing the ML models, thus removes the "single-node" bottleneck on the driver. Figure 12(c) further shows that PS2 can train LDA on billions of documents thanks to the power of data processing in Spark and efficient model management of parameter server.

6.4 Evaluation for Scalability

We report the scalability of PS2 using LR as an example. Specifically, we examine: (1) How would PS2 perform when number of workers/servers increases? (2) How would PS2 perform when size of the model increases?

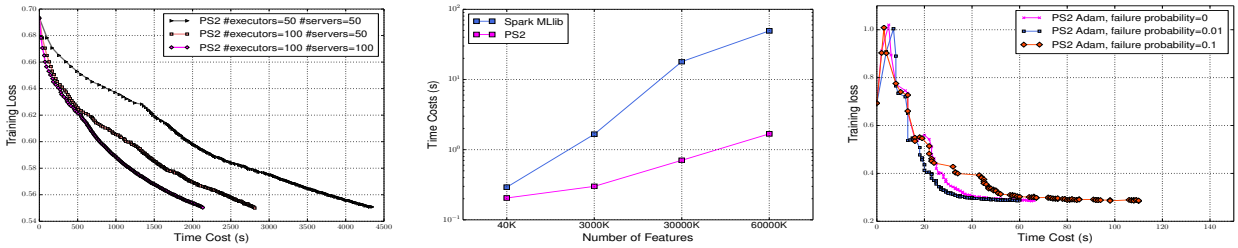
Number of Workers/Servers. Figure 13(a) presents the performance of PS2 when using more computing resources. We test it on CTR dataset with different number of workers and servers. When there are 50 executors and 50 servers, PS2 requires 4519 seconds to reach the objective value of 0.55. When the number of executors increases to 100, the time cost decreases to 2865 seconds. Further, when using 100 executors and 100 servers, the time cost is 2199 seconds. We can see that either increasing number of executors or servers can boost the convergence. When there are more executors, the computation cost on each worker decreases; when more servers are involved, the communication cost on each server is reduced. In addition, PS2 gets 2.05× faster when we double the number of workers and servers, which is a bit more than a linear speedup. We dig into the log and find that, when resources are not enough (i.e., using 50 workers and 50 servers), they are many network failures, thus slowing down the convergence. When given more resources, there are nearly no failures. This experiment demonstrates that PS2 can obtain a good scalability.

Size of Model. We also present the scalability of PS2 with respect to model size. We use 20 executors and 20 servers to train LR on datasets with different number of features, ranging from 40K to 60,000K. With a reference point, we also include the scalability of Spark MLlib. The results are shown in Figure 13(b). We can find the performance of Spark MLlib decreases by 168× when the number of features increases from $40K$ to $60,000K$. In contrast, the time cost per iteration



(a) PS2 vs Petuum/Glint, K=1000 (PubMED) (b) PS2 vs Spark MLlib, K=100 (PubMED) (c) PS2, K=1000 (App)

Figure 12: Comparison of PS2 with other systems over LDA. We compare PS2 with Spark MLlib for K=100 since Spark MLlib runs out of memory for a large value.



(a) Scalability over #Workers/Servers (CTR) (b) Scalability over Size of Model (c) Fault Tolerance (KDDB)

Figure 13: Evaluation of Scalability and Fault Tolerance.

of PS2 increases by 8.5×, from 0.2 seconds to 1.7 seconds. Thus we conclude the performance of PS2 is highly scalable with respect to model size.

6.5 Evaluation for Fault Tolerance

We demonstrate fault tolerance by task failures, which are much more frequent than worker/server failures in a real-world distributed environment. We simulate task failures by randomly throwing exceptions with a given probability.

Figure 13(c) shows the performance of training LR using 20 workers and 20 servers with probabilities of task failure as 0, 0.01 and 0.1. We can see that the performance gets worse when more task failures happen. To finish the training process, it requires 66 seconds for a normal execution. This number changes to 74 seconds and 127 seconds when the failure probability turns to 0.01 and 0.1. Also, all these three cases can converge to the same solution. This indicates that even in an extreme environment where 10% tasks fail, PS2 can still finish the training process in a reasonable time cost.

7 RELATED WORK

Spark Ecosystem. Spark [32] is a popular distributed dataflow system for big data analytics. In Spark, there are two types of nodes, i.e., one driver and multiple executors. The driver is responsible for scheduling executors and the executors are used for data loading and processing. Resilient Distributed Datasets (RDD) is the core abstraction of Spark, which is a distributed memory that allows programmers to perform in-memory computations on large clusters in a fault-tolerant manner. Spark powers a stack of libraries including Spark SQL [5], GraphX [11] and Spark Streaming [33] for dealing with data from different sources. There are also some other ML systems built on top of Spark. For example, KeyStoneML [24] introduces pipeline optimization for ML tasks with a high-level API and SystemML [6] introduces declarative ML on top of Spark. MLlib* [34] further optimizes MLlib by integrating MLlib with model averaging and AllReduce implementation in the context of generalized linear models.

ML Systems Based on Parameter Servers. Parameter server is a distributed key-value store and widely used for ML. It is often used to manage the ML models for distributed ML systems and provides pull/push operators for workers to access the model parameters. There are many ML systems proposed based on parameter servers. For example, LightLDA [30] uses parameter servers to store the "topic-word" matrix in LDA and accelerate the training process. Petuum [28] and Angel [18] are two general-purpose systems for distributed ML based on parameter servers. They support many ML models like LR, SVM, LDA, etc.

Parameter Servers on Spark. To leverage both the benefits of parameter servers for efficient ML training and Spark for data processing, there are some other systems trying to integrate Spark with parameter servers. Similar as PS2, these systems use parameter servers to store model parameters and Spark for processing training data. Glint [14] is an asynchronous parameter server implementation on Spark for LDA and DistML includes a monitor to manage the parameter servers and workers. However, these approaches are not robust enough and provide limited primitive interfaces, such as pull/push, which do not consider the diversity of operations on ML models in parameter servers.

Deep Learning Systems. Deep learning is increasingly popular and many systems have been proposed like TensorFlow [4] and MXNet [7]. To leverage the power of Spark for efficient data processing, there are also some efforts to integrate Spark with these deep learning sytems. Notable examples include BigDL [9], TensorFlowOnSpark [2], CaffeOnSpark [1], MMLSpark [13], etc. In this paper, we focus on using parameter servers to address the bottleneck of Spark, in dealing with non-deep learning large models like LR, GBDT, LDA and graph embedding. We leave the deep learning part as future work.

8 CONCLUSIONS

In this paper, we proposed PS2 to bring Spark the power of training large-scale ML models with high efficiency. PS2 uses Spark for efficient data processing and parameter servers for distributed model management. The modification of PS2 over Spark is transparent to Spark users, and does not hack the core of Spark, by launching Spark and parameter server as two separate applications. By analyzing operations on ML model parameters in Tencent workloads, we identified that simple pull/push operators are not enough. We further extended PS2 with DCV, with a rich set of operators for operating ML models. Experimental results on both public and Tencent workloads show that PS2 can be up to 55.6× faster than baseline systems. PS2 has now been deployed in Tencent to accelerate ML tasks for many real-world applications.

9 ACKNOWLEDGEMENT

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

REFERENCES

[1] Caffeonspark. https://github.com/yahoo/CaffeOnSpark.
[2] Tensorflowonspark. https://github.com/yahoo/TensorFlowOnSpark.
[3] D. J. Abadi, S. Madden, and N. Hachem. Column-stores vs. row-stores: how different are they really? In *SIGMOD*, pages 967–980, 2008.
[4] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. A. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. Tensorflow: A system for large-scale machine learning. In *OSDI*, pages 265–283, 2016.
[5] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark SQL: relational data processing in spark. In *SIGMOD*, pages 1383–1394, 2015.
[6] M. Boehm, M. Dusenberry, D. Eriksson, A. V. Evfimievski, F. M. Manshadi, N. Pansare, B. Reinwald, F. Reiss, P. Sen, A. Surve, and S. Tatikonda. Systemml: Declarative machine learning on spark. *PVLDB*, 9(13):1425–1436, 2016.
[7] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *CoRR*, abs/1512.01274, 2015.
[8] W. Chen, Z. Wang, and J. Zhou. Large-scale L-BFGS using mapreduce. In *Annual Conference on Neural Information Processing Systems*, pages 1332–1340, 2014.
[9] J. Dai, Y. Wang, X. Qiu, D. Ding, Y. Zhang, Y. Wang, X. Jia, C. Zhang, Y. Wan, Z. Li, J. Wang, S. Huang, Z. Wu, Y. Wang, Y. Yang, B. She, D. Shi, Q. Lu, K. Huang, and G. Song. Bigdl: A distributed deep learning framework for big data. *CoRR*, abs/1804.05839, 2018.
[10] J. C. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *JMLR*, 12:2121–2159, 2011.
[11] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. Graphx: Graph processing in a distributed dataflow framework. In *OSDI*, pages 599–613, 2014.
[12] A. Grover and J. Leskovec. node2vec: Scalable feature learning for networks. In *SIGKDD*, pages 855–864, 2016.
[13] M. Hamilton, S. Raghunathan, A. Annavajhala, D. Kirsanov, E. de Leon, E. Barzilay, I. Matiach, J. Davison, M. Busch, M. Oprescu, R. Sur, R. Astala, T. Wen, and C. Park. Flexible and scalable deep learning with mmlspark. *CoRR*, abs/1804.04031, 2018.
[14] R. Jagerman, C. Eickhoff, and M. de Rijke. Computing web-scale topic models using an asynchronous parameter server. In *SIGIR*. ACM, 2017.
[15] J. Jiang, B. Cui, C. Zhang, and F. Fu. Dimboost: Boosting gradient boosting decision tree to higher dimensions. In *SIGMOD*, pages 1363–1376, 2018.
[16] J. Jiang, B. Cui, C. Zhang, and L. Yu. Heterogeneity-aware distributed parameter servers. In *SIGMOD*, pages 463–478, 2017.
[17] J. Jiang, J. Jiang, B. Cui, and C. Zhang. Tencentboost: A gradient boosting tree system with parameter server. In *ICDE*, pages 281–284, 2017.
[18] J. Jiang, L. Yu, J. Jiang, Y. Liu, and B. Cui. Angel: a new large-scale machine learning system. *National Science Review*, pages 216–236, 2017.
[19] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.
[20] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *NIPS*, pages 1106–1114, 2012.
[21] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su. Scaling distributed machine learning with the parameter server. In *Proc. OSDI*, pages 583–598, 2014.

[22] X. Meng, J. K. Bradley, B. Yavuz, E. R. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. B. Tsai, M. Amde, S. Owen, D. Xin, R. Xin, M. J. Franklin, R. Zadeh, M. Zaharia, and A. Talwalkar. Mllib: Machine learning in apache spark. *JMLR*, 17:34:1–34:7, 2016.

[23] B. Perozzi, R. Al-Rfou, and S. Skiena. Deepwalk: online learning of social representations. In *SIGKDD*, pages 701–710, 2014.

[24] E. R. Sparks, S. Venkataraman, T. Kaftan, M. J. Franklin, and B. Recht. Keystoneml: Optimizing pipelines for large-scale advanced analytics. In *ICDE*, pages 535–546, 2017.

[25] R. Srivastava, G. K. Palshikar, S. Chaurasia, and A. Dixit. What's next? a recommendation system for industrial training. *Data Science and Engineering*, pages 232–247, 2018.

[26] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[27] J. Tang, M. Qu, M. Wang, M. Zhang, J. Yan, and Q. Mei. LINE: large-scale information network embedding. In *WWW*, pages 1067–1077, 2015.

[28] E. P. Xing, Q. Ho, W. Dai, J. K. Kim, J. Wei, S. Lee, X. Zheng, P. Xie, A. Kumar, and Y. Yu. Petuum: A new platform for distributed machine learning on big data. In *SIGKDD*, pages 1335–1344, 2015.

[29] L. Yu, B. Cui, C. Zhang, and Y. Shao. Lda*: A robust and large-scale topic modeling system. *PVLDB*, 10(11):1406–1417, 2017.

[30] J. Yuan, F. Gao, Q. Ho, W. Dai, J. Wei, X. Zheng, E. P. Xing, T.-Y. Liu, and W.-Y. Ma. Lightlda: Big topic models on modest computer clusters. In *WWW*, pages 1351–1361. ACM, 2015.

[31] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, pages 15–28, 2012.

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

[33] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: fault-tolerant streaming computation at scale. In *SOSP*, pages 423–438, 2013.

[34] Z. Zhang, J. Jiang, W. Wu, C. Zhang, L. Yu, and B. Cui. Mllib*: Fast training of glms using spark mllib. In *ICDE*, 2019.

A HYPERPARAMETER SETTING

The hyperparameters for each workload used in the paper are presented in Table 4.

Model	Hyperparameters
LR	learning_rate = 0.618 mini_batch_fraction = 0.01 $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 1e-8$
DeepWalk	length_of_random_walk = 8 batch_size = 512, learning_rate = 0.01 window_size = 4, negative_sampling = 5
GBDT	learning_rate = 0.1 number_of_trees = 100 max_depth = 7 size_of_histogram = 100
LDA	$\alpha = 0.5$, $\beta = 0.01$

Table 4: Settings of hyperparameters used in this paper.