DimBoost: Boosting Gradient Boosting Decision Tree to Higher Dimensions

Jiawei Jiang[†§] Bin Cui[†] Ce Zhang[‡] Fangcheng Fu[†]

[†]School of EECS & Key Laboratory of High Confidence Software Technologies (MOE), Peking University
[‡]Department of Computer Science, ETH Zürich [§]Tencent Inc.
[†]{blue.jwjiang, bin.cui, ccchengff}@pku.edu.cn [‡]ce.zhang@inf.ethz.ch [§]jeremyjiang@tencent.com

ABSTRACT

Gradient boosting decision tree (GBDT) is one of the most popular machine learning models widely used in both academia and industry. Although GBDT has been widely supported by existing systems such as XGBoost, LightGBM, and MLlib, one system bottleneck appears when the dimensionality of the data becomes high. As a result, when we tried to support our industrial partner on datasets of the dimension up to 330K, we observed suboptimal performance for all these aforementioned systems. In this paper, we ask *"Can we build a scalable GBDT training system whose performance scales better with respect to dimensionality of the data?"*

The first contribution of this paper is a careful investigation of existing systems by developing a performance model with respect to the dimensionality of the data. We find that the *collective communication operations* in many existing systems only implement the algorithm designed for small messages. By just fixing this problem, we are able to speed up these systems by up to 2×. Our second contribution is a series of optimizations to further optimize the performance of collective communications. These optimizations include a task scheduler, a two-phase split finding method, and low-precision gradient histograms. Our third contribution is a sparsity-aware algorithm to build gradient histograms and a novel index structure to build histograms in parallel. We implement these optimizations in DimBoost and show that it can be 2-9× faster than existing systems.

CCS CONCEPTS

• **Computing methodologies** → **Machine learning algorithms**;

KEYWORDS

Gradient Boosting Decision Tree; High-dimensional Feature; Parameter Server; Collective Communication

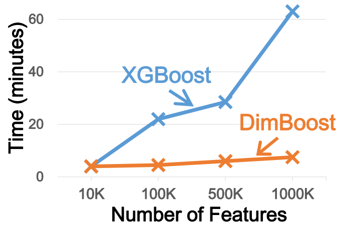


Figure 1: **Performance of XGBoost and DimBoost with respect to # features on one example dataset.**

ACM Reference Format:
Jiawei Jiang, Bin Cui, Ce Zhang, Fangcheng Fu. 2018. DimBoost: Boosting Gradient Boosting Decision Tree to Higher Dimensions. In *SIGMOD'18: 2018 International Conference on Management of Data, June 10–15, 2018, Houston, TX, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3183713.3196892

1 INTRODUCTION

Gradient boosting decision tree (GBDT) is a boosting-based machine learning model that ensembles a set of "weak learners", each of which is a regression tree [15]. It is also one of the most popular choices [7] in data analytics competitions such as Kaggle and KDDCup, and has been widely used in both academia and industry.
(Motivating Application) Our industrial partner, Tencent Inc., trains gradient boosting tree models in one of their production pipelines. The application is to predict the gender of users. The features used for training are fused from multiple data sources. Because they use one-hot encoding for categorical features and use Cartesian product to generate new features as the combination of multiple original features, the number of features (dimensional of data) in the current production pipeline is up to 330K while the total size of the dataset is up to 150 gigabytes.

There are many existing platforms that provide distributed solutions to train GBDT models. Prominent examples include MLlib [29], XGBoost [7], and LightGBM [28]. However, when our industrial partner deployed these systems to their dataset, they observed that the performance decreases significantly with respect to the increase of dimensionality. Figure 1 shows how the performance of XGBoost changes with respect to the dimension of features. Motivated by this, in this paper, we ask the question, "*What are the reasons behind the performance bottlenecks of existing systems for high-dimensional data, and how can we fix it?*".
(Premier of Gradient Boosting Decision Trees) GBDT builds a set of binomial trees from the root node to the leaf node. The essence is selecting a split entry, consisting of a feature f and a

value v, based on which a tree node is split into one left child node and one right child node. We assign the instances whose feature f is less than v to the left child, and those larger than v to the right child. For this purpose, we summarize the gradients of instances to a data structure called the gradient histogram. As we will elaborate in Section 2, we then find the optimal split entry with the gradient histogram. Below, we briefly describe the execution procedure of training GBDT models to set the context for our discussions of our technical contributions. Most distributed implementations of GBDT share the following *core operation* to create a single regression tree.

(1) **Data Partitioning.** Training dataset is partitioned into several shards, each of which is assigned to one worker.
(2) **Gradient Histogram Construction.** Each worker summarizes its local data copy into M *gradient histograms* where M equals to the total number of features.
(3) **Histogram Aggregation and Split Finding.** All workers propose w different gradient histograms for a single feature where w denotes the number of workers. These local histograms are aggregated into a single global histogram (the aggregation function is simply a sum). Then, the system chooses the "split feature" and "split value" from these "global gradient histograms".
(4) **Tree Splitting.** Once receiving the split feature and split value, each worker splits the current tree node and proceeds one layer deeper into the tree.

(Summary of Technical Contributions) When running the above core operation on hundreds of machines, we observed that the communication overhead in step 3 and the computation overhead in step 2 are the major bottlenecks. Not surprisingly, when the datasets contain more features, these bottlenecks become more significant. In this paper, we develop a set of novel optimizations for histogram construction, aggregation, compression, and split decision. These optimizations take advantage of the sparsity of data or use low precision data representation for communication. DimBoost is the prototype that implements all these optimizations. On a production cluster of Tencent, DimBoost is up to 2-9× faster than existing platforms. Our technical contributions can be summarized as follows.

C1: Analysis of Existing Systems. We first conduct a systematic study of existing GBDT systems focusing on the communication operators: MLlib uses the MapReduce [12, 17] framework to aggregate parameters with an all-to-one reduce operator, XGBoost uses `AllReduce` to summarize local solutions by performing a binomial tree reduce, and LightGBM uses `ReduceScatter` to exchange local solutions recursively. Despite of decades of study by the HPC community on implementing these communication operators for *both* large and small messages [34], most GBDT systems implemented their own version of these operators *only using the algorithms designed for small messages*. As a result, not surprisingly, these systems work well on low dimensional data when all messages exchanged are relatively small, but suffer for high dimensional data. In DimBoost, we use an architecture based on parameter server [8, 11] that has a communication overhead matching `ReduceScatter` for large messages [34]. This simple optimization alone improves the system performance by up to 2×. Compared with existing parameter-server systems [1, 10, 36], DimBoost is also the first GBDT system built with the parameter server architecture.

C2: Optimizing for Communication — Efficient Histogram Aggregation and Tree Splitting. Our second contribution is a series of optimizations designed to further decrease the communication overhead. Training GBDT models requires aggregating the gradient histograms of all the features to find the best split. Therefore, the communication cost of histogram aggregation is proportional to the dimension of features. To amortize the linear overhead with respect to the dimensionality, we first design a task scheduler to evenly distribute the task of finding global tree splits among *all* workers to avoid overloading a single master worker. We then compress the gradient histogram with low-precision data representation to decrease the amount of data movement during aggregation. This compression scheme quantizes 32-bit floating-point gradients into 8-bit fixed-points. The tree splitting operation is decoupled into two phases to reduce expensive message passing. These optimizations together improve the performance of DimBoost by up to 3×.

C3: Optimizing for Computation — Sparsity-aware and Gradient Histogram Construction. Our third contribution is to develop an efficient data structure to optimize for the computation overhead of building gradient histograms. Most existing systems implicitly assume that the dataset is dense *during histogram construction*, and therefore enumerate *all* features of each instance. However, when the dimensionality is high, often the dataset is sparse. To optimize for sparse datasets, we design a simple, but novel, data structure to allow the algorithm to only access nonzero features when building a gradient histogram. We also discuss how to build multiple gradient histograms in parallel. These optimizations improve the performance by up to 2×.

C4: System Implementation and Evaluation. We implement all aforementioned optimizations in DimBoost. Despite of the simplicty of these proposed optimizations, to our best knowledge, this is the first time that these optimizations are integrated into a GBDT training system. We compare DimBoost with existing GBDT systems, including XGBoost, LightGBM, TecentBoost, and MLlib, using both public datasets and production datasets provided by Tencent Inc. On clusters with up to 50 machines, DimBoost achieves up to 9× speedup over these existing systems.

Overview. The rest of this paper is organized as follows. We present the preliminaries in Section 2, followed by an analysis of existing GBDT systems in Section 3. We present an overview of DimBoost in Section 4. The optimizations for histogram construction and tree splitting are presented in Section 5 and Section 6, respectively. We present experimental results in Section 7, discuss related work in Section 8, and conclude in Section 9.

2 PRELIMINARIES

2.1 Input Dataset

The input to the training procedure is a set of *instances*: $\{(\mathbf{x}_i, y_i)\}_{i=1}^{N}$ where N is the number of training instances [27]. We call each $\mathbf{x}_i \in \mathbb{R}^M$ the *feature vector* of an instance (M is the *dimensionality* of an instance) and $y_i \in \mathbb{R}$ the *label* of an instance. When the data set is dense, each feature vector $\mathbf{x}_i$ can be stored as a dense array; when the data set is sparse, only nonzero elements need to be stored as a pair of their *index* and corresponding *feature value*.

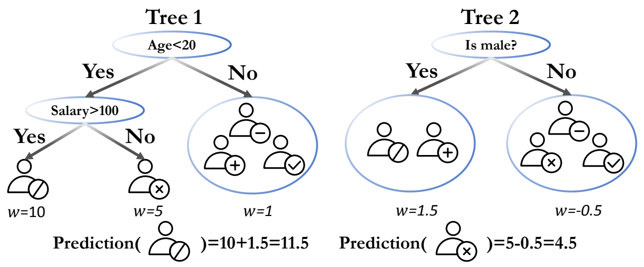


Figure 2: An Example of GBDT.

2.2 Gradient Boosting Decision Trees

Gradient boosting decision trees (GBDT) is one popular *tree ensemble model* [6, 15]. Figure 2 illustrates a GBDT model — In each tree, each training instance $\mathbf{x}_i$ is classified to one leaf node which predicts the instance with a weight ω. GBDT uses the regression tree that predicts a continuous weight on one leaf, rather than the decision tree in which each leaf predicts a class. GBDT sums the predictions of all the trees and gets the prediction for a single instance:

$$\widehat{y}_i = \sum_{t=1}^{T} \eta f_t(\mathbf{x}_i) \quad (1)$$

where T denotes the number of trees, and $f_t(\mathbf{x}_i)$ denotes the prediction of the t-th tree.

Training GBDT Models. Unlike linear and neural models which can be trained with gradient-based optimization methods [4, 5, 40], GBDT is trained in an additive manner. For the t-th tree, we minimize the following regularized objective.

$$F^{(t)} = \sum_{i=1}^{N} l(y_i, \widehat{y}_i^{(t)}) + \Omega(f_t) = \sum_{i=1}^{N} l(y_i, \widehat{y}_i^{(t-1)} + f_t(\mathbf{x}_i)) + \Omega(f_t)$$

where l is a loss function that calculates the loss given the prediction and the target (e.g., logistic loss $l = log(1 + e^{-y_i \widehat{y}_i})$ or square loss $l = (y_i - \widehat{y}_i)^2$). Ω is a regularization function for avoiding over-fitting. In this paper, we follow XGBoost [7] and choose Ω of the following form:

$$\Omega(f_t) = \gamma L + \frac{1}{2}\lambda ||\omega||^2$$

where L is the number of leaves in the tree, and ω is the weight vector. γ and λ are hyper-parameters. LogitBoost [14] expands $F^{(t)}$ via a second-order approximation.

$$F^{(t)} \approx \sum_{i=1}^{N} [l(y_i, \widehat{y}_i^{(t-1)}) + g_i f_t(\mathbf{x}_i) + \frac{1}{2} h_i f_t^2(\mathbf{x}_i)] + \Omega(f_t)$$

where g_i and h_i are the first- and second-order gradients: $g_i = \nabla_{\widehat{y}_i} l(y_i, \widehat{y}_i)$, $h_i = \nabla^2_{\widehat{y}_i} l(y_i, \widehat{y}_i)$. Let $I_j = \{i | x_i \in leaf_j\}$ be the set of instances belonging to the j-th leaf and remove the constant term, we get the approximation of $F^{(t)}$:

$$\widetilde{F}^{(t)} = \sum_{j=1}^{L} [(\sum_{i \in I_j} g_i)\omega_j + \frac{1}{2}(\sum_{i \in I_j} h_i + \lambda)\omega_j^2] + \gamma L$$

The optimal weight and objective of the j-th leaf are:

$$\omega_j^* = -\frac{\sum_{i \in I_j} g_i}{\sum_{i \in I_j} h_i + \lambda}, \quad \widetilde{F}^* = -\frac{1}{2}\sum_{j=1}^{L} \frac{(\sum_{i \in I_j} g_i)^2}{\sum_{i \in I_j} h_i + \lambda} + \gamma L$$

Algorithm 1 Greedy Splitting Algorithm

```
$M$: # features, $N$: # instances, $K$: # split candidates
$g_i$, $h_i$: first-order and second-order gradients of an instance
1: for $m = 1$ to $M$ do
2:     generate $K$ split candidates $S_m = \{s_{m1}, s_{m2}, \ldots, s_{mk}\}$
3: end for
4: for $m = 1$ to $M$ do
5:     loop $N$ instances to generate gradient histogram with $K$ buckets
6:     $G_{mk} = \sum g_i$ where $s_{mk-1} < x_{im} < s_{mk}$
7:     $H_{mk} = \sum h_i$ where $s_{mk-1} < x_{im} < s_{mk}$
8: end for
9: $gain_{max} = 0, G = \sum_{i=1}^N g_i, H = \sum_{i=1}^N h_i$
10: for $m = 1$ to $M$ do
11:     $G_L = 0, H_L = 0$
12:     for $k = 1$ to $K$ do
13:         $G_L = G_L + G_{mk}, H_L = H_L + H_{mk}$
14:         $G_R = G - G_L, H_R = H - H_L$
15:         $gain_{max} = max(gain_{max}, \frac{G_L^2}{H_L+\lambda} + \frac{G_R^2}{H_R+\lambda} - \frac{G^2}{H+\lambda})$
16:     end for
17: end for
18: Output the split with max gain
```

We can enumerate every possible tree structure to find the optimal solution. However, this scheme is arguably impractical in practice. Therefore, existing research works generally adopt a greedy algorithm to successively split tree nodes, as illustrated in Algorithm 1. Given K split candidates for each feature at a given node, we enumerate all the instances belonging to this node to build a gradient histogram that summaries the gradient statistics (line 4-8). Specially, G_{mk} sums the first-order gradients of the instances whose m-th features fall into the range of the $(k-1)$-th and the k-th split candidates, and H_{mk} sums the second-order gradients in the same manner. After building the gradient histograms for all the features, we use the histograms to find the split that gives the maximal objective gain (line 10-17) defined as

$$Gain = \frac{1}{2}[\frac{(\sum_{i \in I_L} g_i)^2}{\sum_{i \in I_L} h_i + \lambda} + \frac{(\sum_{i \in I_R} g_i)^2}{\sum_{i \in I_R} h_i + \lambda} - \frac{(\sum_{i \in I} g_i)^2}{\sum_{i \in I} h_i + \lambda}] - \gamma$$

where I_L and I_R are the left and the right child nodes after splitting. Note that we can propose split candidates with several methods. The exact method sorts all the instances by each feature and uses all possible splits. When the exact method is too time-consuming, previous work uses percentiles of feature distribution [7, 28]. Many works have designed distributed quantile sketch algorithms for this purpose, such as GK [18], DataSketches [37], and WQS [7].

Two prevailing techniques used to prevent over-fitting are shrinkage and feature sampling. Shrinkage multiplies the leaf weight in Eq.(1) by a hyper-parameter η called learning rate before adding it to the prediction. Feature sampling technique samples a subset of features for each tree. It has been proved to be effective in practice [7] in improving both performance and generalization.

2.3 Implementations of Existing Systems

MLlib. There are two types of nodes in MLlib [29], i.e., master (driver) and worker (executor). MLlib partitions the training data by instances (rows) over a set of workers. The training process is presented as follows: 1) In each iteration, the master node pulls off active tree nodes from a waiting queue. 2) The algorithm splits this set of tree nodes. In order to choose the best split for a given tree node, data statistics (gradients) are collected from distributed workers. 3) For each tree node, the statistics are collected to a particular worker node via a `reduceByKey` operator. 4) The designated worker chooses the best split result (feature-value pair), or chooses to stop splitting if the stopping criteria are met. 5) The master collects all decisions about splitting nodes and updates the model. 6) The updated model is passed to the workers on the next iteration. 7) This process iterates until the queue of active tree node is empty. In MLlib, statistics aggregation is the bottleneck. In general, this implementation is bound by either the cost of statistics computation on workers or by communicating the statistics.

XGBoost. In XGBoost [7], the first two steps are the same as MLlib that each worker calculates the data statistics (gradients). But different from MLlib, there is no master node in XGBoost. To perform the aggregation operation, XGBoost uses the `AllReduce` operator. All workers are organized as a binomial tree consisting of leaf workers, internal workers, and a root worker. The statistics of training data are communicated following the bottom-up path of this tree structure. Specifically, the aggregation starts at the leaf worker. In the first communication step, the statistics of two leaf workers are merged on the parent worker. This process repeats in the same manner until reaching the root worker. Once the root worker gets the merged statistics, it calculates the split result and updates the model. The updated model is sent to all the workers via an up-bottom strategy which starts at the root worker and ends at the leaf workers. Then, the next split iteration begins.

LightGBM. LightGBM [28] provides both data-parallel and feature-parallel implementations for GBDT. The data-parallel strategy partitions the training data by instances (rows), while the feature-parallel strategy partitions the training data by features (columns). However, the feature-parallel implementation of LightGBM needs to load the whole dataset on every worker, which is impractical for many large-scale datasets. The data-parallel strategy operates almost the same as XGBoost, except for the aggregation stage. Instead of merging statistics via a binomial tree, LightGBM uses the `ReduceScatter` operator. In `ReduceScatter`, each worker is responsible for merging a part of the statistics. Briefly speaking, LightGBM uses a divide-and-exchange strategy. LightGBM sorts all workers as a list. In each communication step, LightGBM equally divides the list into two sublists and exchanges necessary statistics between two sublists. The next communication step further divides two sublists. Since the size of a sublist is halved in each step, the communication cost is also halved.

3 ANALYSIS OF EXISTING GBDT SYSTEMS

One of the largest system bottleneck for existing GBDT systems on high dimensional data is the communication operator that simply calculates the `sum` of all local histograms. We now analyze the different model aggregation approaches adopted by existing GBDT systems and compare their performance.

Problem Definition. Following Thakur et al. [34], we use a cost model to analyze the cost of an aggregation method. We assume that there exist w workers and the size of a local gradient histogram is h bytes. We model the time needed for a worker to send or receive a package as $\alpha + n\beta$ where α is the latency for each package, β is the transfer time per byte, and n is the number of bytes transferred via the network. We assume that γ is the computation cost per byte for merging two histograms. Note that, the computation time is often less than the transmission time.

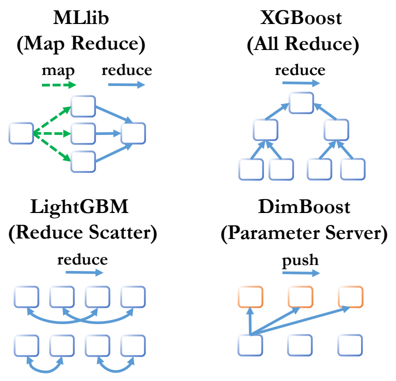


Figure 3: Existing Model Aggregation Methods.

System	# comm steps	communication time
MLlib	1	$h\beta w + \alpha + h\gamma$
XGBoost	$\log w$	$(h\beta + \alpha + h\gamma)\log w$
LightGBM	$\log w$	$\frac{w-1}{w}h\beta + (\alpha + h\gamma)\log w$
DimBoost	1	$\frac{w-1}{w}h\beta + (w-1)\alpha + h\gamma$

Table 1: Comparison of communication cost. w: # workers, h: histogram size, α: latency per package, β: transfer time per byte, γ: merging time per byte.

MLlib. MLlib [29] uses `MapReduce` for model aggregation. As shown in Figure 3, MLlib merges local parameters in the reduce phase. Since the reduce operation is performed by a single node, the coordinator needs to receive hw bytes. In summary, MLlib takes one communication step and in total $(h\beta w + \alpha + h\gamma)$ time.

XGBoost. A popular GBDT engine, XGBoost, uses `AllReduce` by organizing all the workers as a binomial tree. The aggregation follows a bottom-to-up scheme starting from the leaves and ending at the root — $\log w$ steps in total. It is worthy to note that these steps cannot overlap in XGBoost's implementation. Therefore, XGBoost needs $\log w$ communication steps and $(h\beta + \alpha + h\gamma)\log w$ time.

LightGBM. LightGBM uses `ReduceScatter` for model aggregation. `ReduceScatter` is a variant of `Reduce` in which the result, instead of being merged at the root node, is scattered among all workers. As shown in Figure 3, LightGBM implements a recursive-halving strategy that the communication cost is halved at each step. Specifically, in the first step, each worker exchanges $\frac{h}{2}$ data with a worker that is $\frac{w}{2}$-distance away; in the second step, each worker exchanges $\frac{h}{4}$ data with a worker that is $\frac{w}{4}$-distance away. This process iterates until the neighboring workers exchange their data. To sum up, LightGBM needs $\log w$ communication steps and $(\frac{w-1}{w}h\beta + (\alpha + h\gamma)\log w)$ time. However, the above result only applies to a case that w is a power of two. If w is not a power of two, the time taken by LightGBM is doubled.

DimBoost. As we will see in Section 4, our system implements the model aggregation using the parameter server architecture. DimBoost co-locates workers and servers, meaning that there exist one worker and one server on each physical node. Each worker needs to send $(w-1)$ packages in a batch, each of which is $\frac{h}{w}$

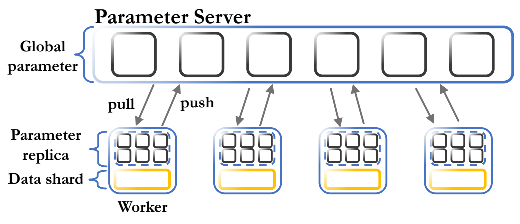


Figure 4: Parameter server architecture

bytes. DimBoost only needs one communication step and takes $(\frac{w-1}{w}h\beta + (w-1)\alpha + h\gamma)$ time. Please note, TencentBoost [22] which is also based on parameter server architecture [38], briefly analyzes the communication cost of collective operations. However, it lacks an explicit cost model and a thorough analysis, and ignores the time cost of network latency and computation.

Remarks. We summarize the time cost of four systems in Table 1. In the presence of a large histogram, the item with h dominates the overall time cost. It is obvious that DimBoost and LightGBM outperform the other two with a large w. Therefore, DimBoost and LightGBM are more suitable for a large message and a large cluster. If w is a power of two, they consume comparable time. Otherwise, LightGBM consumes about twice the time of DimBoost.

A first glance of the result might lead to the conclusion that the parameter server architecture outperforms other model aggregation approaches. However, we note that the communication cost of MLlib, XGBoost, and LightGBM *does not* match the best known communication cost of `MapReduce`, `AllReduce`, and `ReduceScatter` in the literature for large messages [34]. For example, it is known that the bionomial tree algorithm for `AllReduce` is good for small messages but the Rabenseifner algorithm is better for large messages [34]. By choosing a model aggregation strategy that is designed for large messages, DimBoost outperforms existing systems.

4 SYSTEM OVERVIEW

We present a system overview as illustrated in Figure 5.

4.1 Parameter Server Architecture

DimBoost uses the parameter server (PS) architecture [25] as illustrated in Figure 4, which has been shown to be a promising candidate to address the challenge of aggregating a high dimensional parameter [36]. In a parameter server architecture, several machines together store a parameter to prevent the single-point bottleneck, and provide interfaces for workers to *push* and *pull* parameters. Each worker holds a local copy of the parameter, and periodically pushes parameter updates to the PS.

4.2 System Roles

There are three types of roles in DimBoost— server, worker, and master, as elaborated below.

Server. The parameter servers together store a global copy of a model, while each server stores a model shard. Servers receive local updates from workers and materialize them on the model shard with a user-defined *push* interface. When receiving queries from workers, each server sends the global model through a user-defined *pull* interface.

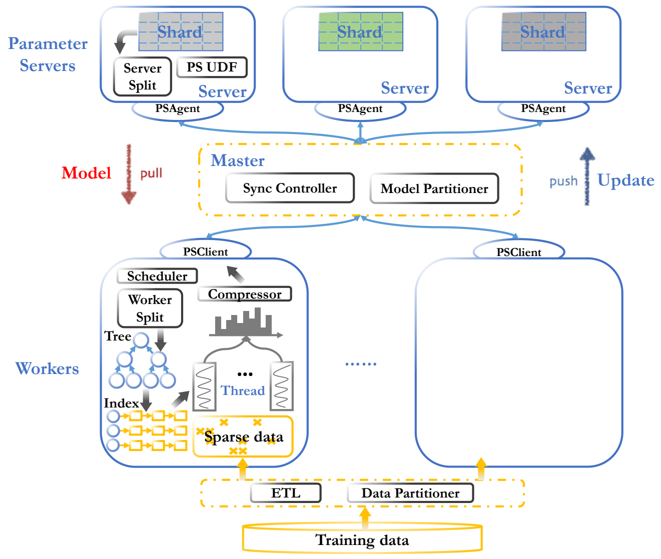


Figure 5: System Overview.

Worker. The training dataset is stored on each worker after ETL (Extract-Transform-Load) and partitioning. Each worker pulls models from servers, and pushes updates.

Master. The master supervises workers and servers with periodical health checking. It also controls the synchronization between workers to assure algorithmic correctness, and manages metadata of model shards.

4.3 Parameter Management of Server

Model parameters are represented as vectors in our system. We use several vectors to represent a matrix-type parameter. For a dense parameter, we store the values of each dimension. For a sparse parameter, we store the ordered indexes and the corresponding values of nonzero elements to reduce the memory cost.

Parameter Partition. To adopt the parameter server (PS) architecture we need to design the strategy of partitioning a parameter vector over several machines. The problem of data partitioning across a set of nodes has been extensively studied by the database community [30], for both centralized [2] and decentralized systems [9]. The most widely used approaches are round-robin partition (successive partition), range partition (value-based partition), and hash partition (partition with a hash function). Hash partition achieves a more balanced query distribution, while range partition facilitates range queries [9]. To achieve a trade-off between query balance and fast range query, we adopt a hybrid range-hash strategy [16]. We first partition a vector to several ranges based on feature indexes, then use hash partition to put each partition onto one node. Users can set the number of partitions according to their specific scenarios. The default number of partitions is the number of parameter servers.

Parameter Manipulation. We provide user-defined functions to manipulate a parameter — *push* and *pull*. The default *push* function adds updates to the parameter, and the default *pull* function returns the current parameter.

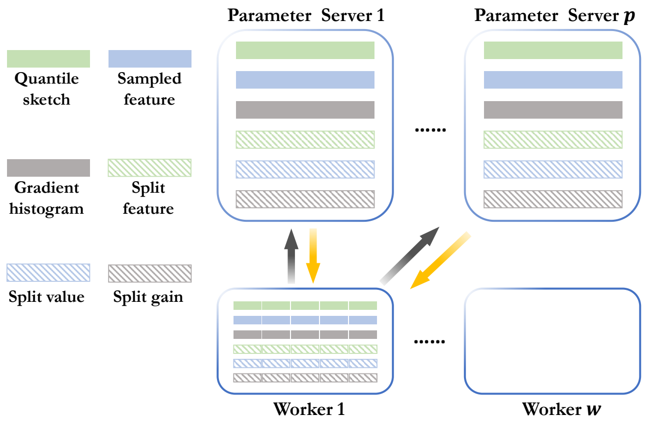


Figure 6: Parameter layout.

Parameter Layout. Figure 6 illustrates the data stored on the parameter server. There are six primary parameters on the PS — the quantile sketches *QtSk*, the sampled features *SpFeat*, the gradient histograms *GradHist*, the split features *SpFeat*, the split values *SpVal*, and the split gains *SpGain*. Unless otherwise stated, the default partition number is equal to p (# parameter servers).

- The data sketches of all features are stored in *QtSk*.
- *SpFeat* stores the sampled features used in each tree.
- *SpFeat*, *SpVal*, and *SpGain* store the split results.
- *GradHist* contains $(2^d - 1)$ rows where d is the maximal tree depth. Each row stores the gradient histogram of a tree node. The size of each row is $2KM\sigma$ where K is the number of split candidates, M is the number of features, and σ is the feature sampling ratio.

4.4 Execution of Worker

The training procedure on each worker contains seven phases, as Figure 7 shows. Among all workers, a leader worker is designated to perform some specific work for coordination.

(1) *CREATE_SKETCH:* Each worker uses its data shard to generate local quantile sketches and pushes them to the PS.
(2) *PULL_SKETCH:* Each worker pulls merged sketches from the PS and proposes split candidates for each feature.
(3) *NEW_TREE:* Each worker creates a new tree and performs some initial work, e.g., initializing the tree structure, computing gradients, and setting the root node to active. The leader worker samples a subset of features.
(4) *BUILD_HISTOGRAM:* Workers use their assigned data to build gradient histograms for active tree nodes. If the root tree node is active, they should first pull the sampled features from the PS.
(5) *FIND_SPLIT:* Each worker pushes local histograms to the PS. The parameter servers add received local histograms to the global one. According to the scheduling results of the task scheduler, each worker pulls the merged gradient histograms of the responsible tree nodes, finds the best split results, and pushes them to the PS.
(6) *SPLIT_TREE:* Each worker 1) pulls split results from the PS, 2) splits active tree nodes, 3) adds children nodes to the tree, 4) deactivates active nodes, and 5) updates the node-to-instance index. If the depth of the tree is smaller than the maximum allowed, the worker sets the children nodes to active and goes to the *BUILD_HISTOGRAM* phase. Otherwise, the worker updates the prediction of each instance, and proceeds to the next tree. Once all trees are finished, the worker goes to the *FINISH* phase.
(7) *FINISH:* The leader worker outputs the trained model.

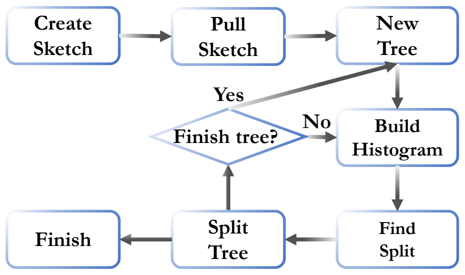


Figure 7: Execution of Worker.

To ensure that different workers proceed in the same pace, we introduce a synchronization barrier that one worker cannot proceed until all workers have finished the current phase. Moreover, we use a layer-wise scheme to consecutively add active nodes [35] — after splitting the current layer, we set the tree nodes of the next layer to active and continue to split the next layer.

4.5 System Bottlenecks and Solutions

We propose a series of optimizations for two critical phases, *BUILD_HISTOGRAM* and *FIND_SPLIT*, briefly summarized below and described in Section 5 and Section 6.

4.5.1 Building Histogram. A **sparsity-aware algorithm** is proposed to build the gradient histogram of a single tree node with its data shard. To build the gradient histograms of several tree nodes in parallel, we devise **a node-to-instance index** with which we can efficiently query the instances of each tree node. Based on the index structure, we further propose **a parallel batch training method** to accelerate the construction of one single histogram.

4.5.2 Finding Split. Each local histogram is transformed to **a low-precision histogram** by a fixed-point compressor before sending to the parameter server. Once local histograms are merged on the parameter servers, **a task scheduler** assigns active tree nodes among all the workers. This method is able to prevent overwhelming a single worker. After receiving assigned tree nodes, each worker calculates the best split for each tree node via **a two-phase split** method. With this approach, the transferred message is significantly reduced.

5 HISTOGRAM CONSTRUCTION

There are two issues in the *BUILD_HISTOGRAM* phase: (1) *how to efficiently build one gradient histogram?* and (2) *how to concurrently build independent gradient histograms?* We describe two optimizations that DimBoost uses.

5.1 Sparse Histogram Construction

The traditional algorithm of building gradient histogram needs to scan all features of a training instance. This training paradigm,

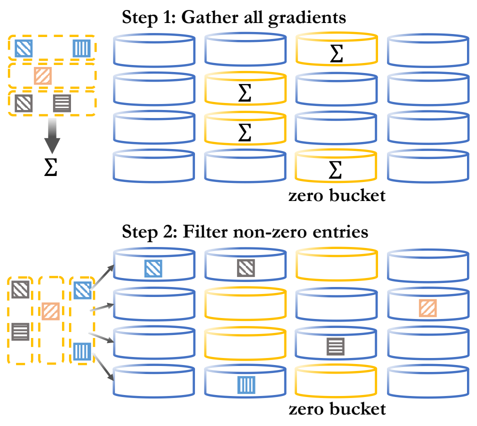


Figure 8: Sparsity-aware Histogram Construction.

designed for a low dimensional dataset, however, incurs a significant cost for a high dimensional dataset. In this section, we study how to reduce the time of building gradient histogram with high-dimensional features.

Observation. We find that a major difference between a low dimensional dataset and a high dimensional one is the data sparsity. High dimensional datasets are generally sparse, that is, the values of most features are zero. With this sparse setting, it is natural to revisit the question that *is it really necessary to access all the zero features?* If we can avoid or batch redundant accesses of zero features, we should be able to decrease the computation time to a large extent via only accessing those nonzero features.

Existing systems already take advantage of sparsity. In XGBoost, it introduces a sparsity-aware method to find split from a gradient histogram. However, to our best knowledge, DimBoost is the first system that utilizes data sparsity to accelerate the construction of gradient histograms.

Note that the sparse representation methods used by other ML algorithms cannot be directly applied to solve the sparsity problem challenging GBDT. For instance, the sparse representation of data matrices is widely used in large-scale generalized linear models [31]. They save space and accelerate training by accessing sparse indexes and ignoring zero features. In our implementation of GBDT, we also store and read data matrix in the sparse format. However, the construction of gradient histograms includes both zero and nonzero features. Our goal is to batch the access of zero features.

Intuition. The intuition is to treat a zero feature as the normal/default case — We assume that all features are zero before actually accessing a dataset. Therefore, we can sum the gradients of training instances and put the sum into the histogram bucket containing the zero feature. Afterwards, we select those nonzero features and handle them individually.

Approach. Based on this intuition, we propose a sparsity-aware algorithm to build a gradient histogram. As shown in Figure 8, we use two steps to build a gradient histogram.

Step 1. We gather the gradients of all instances and put them into the "zero bucket" of each feature, where "zero bucket" denotes the histogram bucket containing a feature value of zero.

Step 2. We filter nonzero elements from each sparse instance and put them into corresponding buckets.

Algorithm 2 Sparsity-aware Histogram Construction

```
$g_i$, $h_i$                        // 1st and 2nd gradients of i-th instance
$sum_g$, $sum_h$                    // the sum of 1st and 2nd gradients
$idx_0(f)$                          // index of feature $f$'s zero histogram bucket
$hist_g(f)(j)$, $hist_h(f)(j)$      // $j$-th histogram bucket of feature $f$
1:  for each instance $x_i$ do
2:      $sum_g \leftarrow sum_g + g_i$
3:      $sum_h \leftarrow sum_h + h_i$
4:      for each nonzero element $(f, v)$ in $x_i$ do
5:          $idx \leftarrow indexOf(f, v)$
6:          $hist_g(f)(idx) \leftarrow hist_g(f)(idx) + g_i$
7:          $hist_h(f)(idx) \leftarrow hist_h(f)(idx) + h_i$
8:          $hist_g(f)(idx_0(f)) \leftarrow hist_g(f)(idx_0(f)) - g_i$
9:          $hist_h(f)(idx_0(f)) \leftarrow hist_h(f)(idx_0(f)) - h_i$
10:     end for
11: end for
12: for each feature $f$ do
13:     $hist_g(f)(idx_0(f)) \leftarrow hist_g(f)(idx_0(f)) + sum_g$
14:     $hist_h(f)(idx_0(f)) \leftarrow hist_h(f)(idx_0(f)) + sum_h$
15: end for
```

Algorithm 2 shows the implementation. g_i and h_i denote the first- and second-order gradients of the i-th instance. sum_g and sum_h denote the sum of first- and second-order gradients. $idx_0(f)$ denotes the index of "zero bucket" of feature f. $hist_g(f)(j)$ and $hist_h(f)(j)$ denote the j-th first-order bucket and second-order bucket of feature f. We now describe the algorithm in details.

(1) First, we add g_i and h_i of each instance to sum_g and sum_h (line 2-3). By combining the accumulation of gradients with the access of nonzero elements, we can save one extra pass over the data. Then, we enumerate nonzero elements of each instance x_i.

(a) For each nonzero element (f, v), we find the histogram bucket that contains value v, denoted by idx. (line 5)

(b) We add g_i and h_i to the corresponding histogram buckets. (line 6-7)

(c) We subtract g_i and h_i from zero buckets. (line 8-9)

(2) After looping over the instances, we conduct a loop over the features and add the sum of gradients, sum_g and sum_h, to the "zero buckets". (line 12-15)

Analysis of Time Complexity. Assuming a dataset has N instances, each of which contains M features, the time complexity of the traditional algorithm is $O(MN)$. Our proposed sparsity-aware algorithm, however, achieves a time complexity of $O(zN + M)$ where z denotes the average number of nonzero elements in an instance. Since $z << M$ in most cases, we significantly decrease the computation cost.

5.2 Parallel Histogram Construction

During the histogram construction, two tree nodes can be processed in parallel if they have no parent-child relationship. Since we train GBDT in a layer-wise manner, we can concurrently process the tree nodes within a layer. We describe our parallel strategy in this section, including a node-to-instance index structure and a parallel batch training method.

Node-to-instance Index. As several threads can build the gradient histograms of different tree nodes in parallel, they need to read the dataset simultaneously. To avoid unnecessary access of

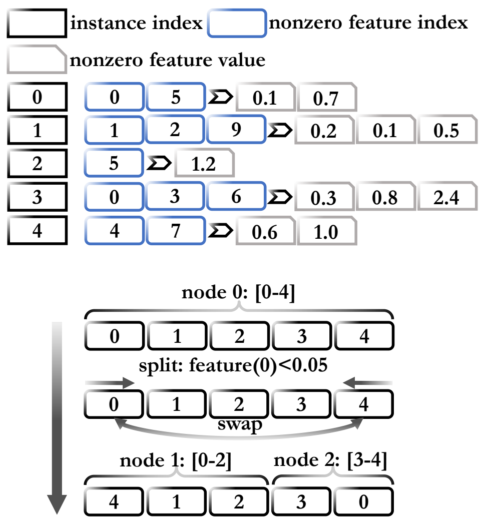


Figure 9: Node-to-instance Index.

the dataset, we design an index structure mapping tree nodes to instances. Figure 9 shows our instance-to-node index. The representation of a training instance is the same as a model parameter, as described in Section 4.3. Since most high dimensional datasets are sparse, Figure 9 chooses the sparse format as an example.

(1) We use an array to store the indexes of all the instances, e.g., $(0, 1, 2, 3, 4)$ in Figure 9.
(2) Each tree node has a range in the index array. At the beginning, all the instances belong to the 0-th node, i.e., the root node. We thereby define that the range of the 0-th node "zero to four".
(3) Given the split result of the root node (the first feature $<$ 0.05), our goal is to put the instances of the 1*st* node to the left of the array and those of the 2*nd* node to the right.
(4) We scan the array from two directions and swap instances in wrong places according to the split result, e.g., swap the 0-th instance and the 4-th instance.
(5) We update the range of two child nodes.

With this index structure, each thread is able to directly find the corresponding instances, rather than scanning the whole dataset.

Parallel Batch Construction. The above node-parallel scheme works in the presence of many tree nodes. However, since there are not so many tree nodes in the first few layers, each thread still needs a long time to sequentially access many instances, which we call the *"cold-start" problem*. To address this problem, we propose a parallel batch training method. From the node-to-instance index, we can acquire the range of the instances of a tree node. Our method divides a range into batches and processes these batches in parallel. Using b to denote the batch size, we build the gradient histogram of a tree node as follows:

(1) Query the node-to-instance index and get the range of the node — $[l, r]$.
(2) Divide the range by b to get $\lceil \frac{r-l+1}{b} \rceil$ batches.
(3) Start $\lceil \frac{r-l+1}{b} \rceil$ threads to process these batches.
(4) Send the gradient histogram to the PS once all the threads are finished.

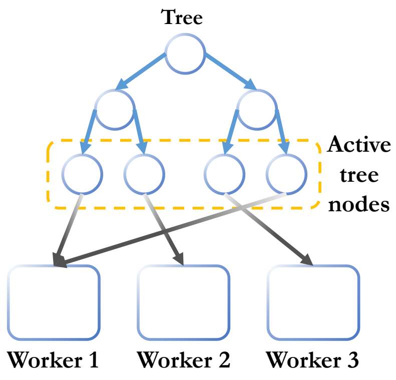


Figure 10: Task Scheduler

6 FINDING SPLIT

Another system bottleneck, other than the construction of gradient histograms, is the *FIND_SPLIT* phase. Since each gradient histogram is partitioned over several parameter servers, we need to design a mechanism to find the best split from distributed partitions. In Figure 7, the leader worker pulls the gradient histogram and finds the best split. However, a gradient histogram can be very large for high dimensional features, and the communication cost is proportional to the number of active nodes. As the tree goes deeper, the number of active tree nodes increases as well — at most 2^{d-1}. To speed up the *FIND_SPLIT* phase, we propose three optimizations.

6.1 Low-precision Gradient Histogram

The first step in the *FIND_SPLIT* phase is to merge local histograms. To reduce the communication cost of pushing local gradient histograms to PS, one intuition is that a floating-point gradient histogram can tolerate some precision loss. Similar ideas have been proved to be effective for linear models in practice, such as ZipML and Difacto [3, 26, 39], yet without implementation, discussion, and theoretical analysis for tree-based GBDT.

In our system, we design a low-precision compressor to compress a gradient histogram. Specifically, for each item q in a histogram, we encode q to a d-bit integer $q' = \lfloor \frac{q}{|c|} \times 2^d \rfloor$ where c denotes the item that has the maximal absolute value in the histogram. To make sure that the expectation of q remains the same after quantification, we add a Bernoulli random number, that is, $q' = \lfloor \frac{q}{|c|} \times 2^d \rfloor + \phi$. Afterwards, the compressed integers and c are sent to the PS. On the PS, we decode the compressed integer q' to a floating-point $q'' = \frac{q'}{2^d} \times |c|$. By adopting this low-precision protocol, we achieve a compression ratio of $\frac{32}{d}$ — a larger d yields a higher precision, while a smaller d brings a higher compression. In practice, we find that $d = 8$ is often enough to obtain no loss on final accuracy. In Appendix A.1, we theoretically prove that this low-precision method can get an objective gain that has the same expectation statistically.

6.2 Round-robin Task Scheduler

After merging local histograms on the PS, the system needs to assign the tasks of splitting active nodes to the workers. The most naive approach is to appoint one worker as an agent to handle all the active nodes. However, this method will incur significant pressure

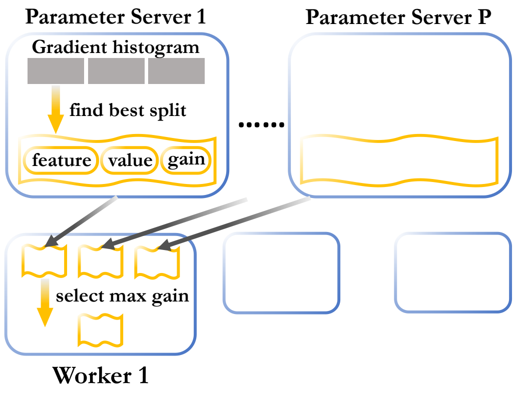


Figure 11: Two-phase Split Finding.

on the agent. Rather than putting the work of finding split on one worker, we adopt a round-robin scheme to schedule the splitting tasks among the workers, as shown in Figure 10. Each worker uses a "state array" to store the "state" of each tree node, where the ($2i+1$)-th item and the ($2i+2$)-th item are the child nodes of the i-th item. Each worker scans this state array and finds responsible active nodes according to a round-robin strategy. For example, the i-th active tree node is assigned to the (i *mod* w)-th worker where w is the number of workers. Then the worker pulls their gradient histograms from the PS and performs the operation of finding split.

6.3 Two-phase Split Finding

Once receiving a responsible tree node from the task scheduler, the worker proceeds to find the best split. Since the gradient histogram is partitioned into several shards, one question is, *how to find the best split from distributed histogram shards?* The most straightforward way is to pull all the shards of a gradient histogram and perform the splitting operation on the worker. Nevertheless, the communication cost is nontrivial. To decrease the communication overhead, we design an approach to decouple the operation of finding split into two phases — the server-side split and the worker-side split.

In Figure 11, when a worker needs to find the best split of a tree node, it operates as follows.

(1) Pull request. The worker sends p requests to p servers to get the shards of a global gradient histogram.
(2) Server-side split. As described in Section 4.3, each parameter server stores a shard of the gradient histogram which contains the gradient summaries of a subset of features. When receiving a pull request, the server finds the optimal split result from its local histogram shard. Then, the split feature, the split value, and the objective gain are sent to the worker.
(3) Worker-side split. Once receiving p local optimal splits, the worker selects the one with the maximal objective gain as the global best split.

The set of local optimal candidates contains the global optimal result, therefore a pass over the local optimal candidates can generate the exact result. Existing platforms generally use parameter server as a distributed key-value store. Although some systems [36] provide user-defined *push* function to support different update scenarios, they do not consider the *pull* function supporting diverse

Dataset	# instance	# features	# nonzero	size
RCV1	0.7M	47K	76	1.4GB
Synthesis	50M	100K	100	60GB
Gender	122M	330K	107	145GB

Table 2: Datasets. # instances refers to the number of instances in a dataset. # features refers to the number of features of an instance. # nonzero refers to the average number of nonzero features in the training instances.

query requirements. Our system, however, is able to customize both *push* and *pull* functions. Therefore, we move the split finding operation of Algorithm 1 (line 10-17) to the *pull* function to implement our two-phase split algorithm. With this method, we reduce the transferred size of a partition to one integer and two floating-point numbers.

7 EVALUATION

7.1 Experimental Setup

System Implementation. DimBoost is implemented in Java and deployed on Yarn. We use Netty [19] to manage the message passing between physical machines. We store the training data on HDFS and design a module to partition the training data. In addition, we provide an easy-to-use data-reading API with memory, disk, and memory-and-disk storage levels. We implement DataSketches to generate quantile sketches. DimBoost has been used in many applications of Tencent for months [21].

Datasets. As shown in Table 2, we use three datasets in our experiments. *RCV1* is a public dataset for text categorization [24]. It contains 700K instances and 47K features. *Synthesis* is a synthesis dataset consisting of 50 million instances and 100K features. *Gender* is a real dataset of Tencent, which is used to predict a user's gender. *Gender* has 122 million instances and 330K features. In some experiments, in order to evaluate the impact of feature dimension, we use partial features of *Gender*. For instance, *Gender*-10K refers to a subset of the first 10K features.

Baselines. We compare DimBoost with four popular systems — MLlib, XGBoost, LightGBM, and TencentBoost [20, 22, 23]. They cover the spectrum of the existing parameter aggregation approaches.

Clusters. Two clusters are used to run these systems. *Cluster-1* is a five-node cluster, in which each machine is equipped with 32GB RAM, 4 cores, and 1GB Ethernet. *Cluster-2* is a productive Yarn cluster consisting of 50 machines, each of which has 64GB RAM, 24 cores, and 1GB Ethernet. To meet the requirement of industrial environments, many machine learning platforms, including MLlib, XGBoost and TencentBoost, use Yarn and HDFS to deploy distributed ML tasks. However, LightGBM needs a sophisticated and repetitive manual configuration on every machine, making it impractical for real applications. Besides, since the evaluated clusters are shared by many users, the maximal memory allocated for each task is limited. Cluster-1 provisions 15GB for each specific task on each machine, and Cluster-2 provisions 10GB. In our experiments, we run five systems in Cluster-1. Afterward, we run them in Cluster-2 except LightGBM since it fails to support our production environment.

Protocol. We use five workers in Cluster-1 and fifty workers in Cluster-2, respectively. For a parameter server system, we use

Metric	Sparsity-aware Construction	Parallel Batch Construction	Node-to-instance Index	Task Scheduler	Two-phase Split	Low-precision Histogram	Time (seconds)
Build the root node	✘	✘	✘	✘	✘	✘	52272
	✔	✘	✘	✘	✘	✘	33
	✔	✔	✘	✘	✘	✘	0.218
Build the last layer	✔	✔	✘	✘	✘	✘	0.806
	✔	✔	✔	✘	✘	✘	0.373
Build a tree	✔	✔	✔	✘	✘	✘	131
	✔	✔	✔	✔	✘	✘	120
	✔	✔	✔	✔	✔	✘	77
	✔	✔	✔	✔	✔	✔	41

Table 3: Effects of Proposed Optimizations.

fifty parameter servers. For the hyper-parameters, we choose $\sigma = 1$ (feature sampling ratio), $T = 20$ (# trees), $d = 7$ (maximal tree depth), $K = 20$ (# split candidates), $b = 10000$ (batch size), $r = 8$ (# compressed bits), $q = 20$ (# threads), and $\eta = 0.01$ (shrinkage learning rate). We split a dataset into two subsets — 90% as the train subset and 10% as the test dataset. The measurements include the end-to-end time to run the train subset, the training error against time, and the predictive accuracy that the trained model performs over the test subset. To guarantee the robustness of the results, we average over three runs for each experiment.

7.2 Effects of Proposed Optimizations

Before comparing DimBoost with other systems, we first run our system to assess the effectiveness of our proposed optimizations. Gender dataset is trained on Cluster-2. We begin with the basic algorithm, consolidate optimizations gradually, and assess how the performance evolves.

Sparsity-aware Histogram Construction. We record the time taken to build the histogram of the root node with and without the sparsity-aware method. The results are presented in Table 3. By applying the sparsity-aware algorithm, the time needed to build a gradient histogram is reduced from 52272 seconds to 33 seconds — a 1500× improvement. The result shows that, by only accessing nonzero features instead of all features, we are able to significantly increase the system speed. Additionally, the performance improvement increases with the increase of feature dimension.

Parallel Batch Histogram Construction. Aimed at solving the "cold-start" problem of the shallow layers, we design a parallel algorithm to build one gradient histogram in batches. Table 3 presents the time taken to build the histogram of the root node. Unsurprisingly, the computation time is reduced from 33 seconds to 218 milliseconds. This significant gain is caused by utilizing the computation capability of a multi-core architecture to concurrently build a gradient histogram.

Node-to-instance Index. We next evaluate the node-to-instance index in the *BUILD_HISTOGRAM* phase. Table 3 shows the time spent to build the last layer of a tree. It is obvious that the use of the index can reduce the computation time from 806 milliseconds to 373 milliseconds— a 2.15× improvement. With this index, several threads can build gradient histograms in parallel without scanning the whole dataset, and therefore a lot of time is saved. This method works better in the presence of more tree nodes.

Task Scheduler. In the *FIND_SPLIT* phase, we establish a task scheduler to assign the splitting tasks among workers. As shown in Table 3, the time needed to train a tree is 120 seconds — 9.2% faster than the way without this method. The performance increase demonstrates that the task scheduler can distribute the burden over all workers to prevent overwhelming a single worker.

Two-phase Split Finding. In addition to the task scheduler, we devise the two-phase split method to reduce the communication cost. The result in Table 3 indicates that the time taken to build a tree is lowered from 120 seconds to 77 seconds. This acceleration is derived from the reduction of transferred message during the *FIND_SPLIT* phase.

Low-precision Gradient Histogram. Finally, we evaluate the effect of the low-precision gradient histogram. In this experiment, we compress a four-bytes floating-point number to an one-byte integer. As Table 3 illustrates, the total time used to build a tree via sending low-precision histograms is 41 seconds, 1.88× faster than choosing the full-precision avenue. An interesting observation is that the predictive accuracy is not significantly damaged after compressing the histograms. Adopting the full-precision transmission method, the test error is 0.2509. With the low-precision avenue, the test error is 0.2514. It verifies the ability of GBDT being tolerable to certain degree of precision loss.

7.3 End-to-end Comparison

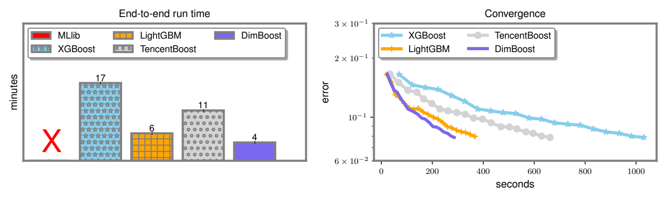
After assessing the techniques we propose, we proceed to compare DimBoost with four competitors. First, we run them in Cluster-1, a small cluster in our lab. Then, we use Cluster-2, a productive cluster.

7.3.1 *Results over a Small Cluster.*

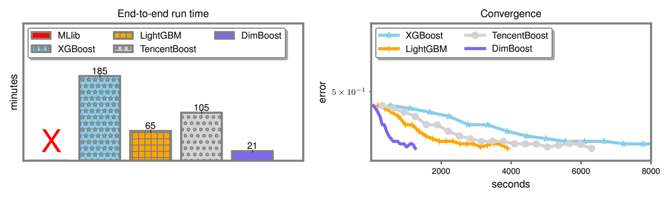
We run five machines to train the *Synthesis* dataset and the *RCV1* dataset. The results are reported in Figure 12, including the end-to-end run time and the evaluation metric with respect to the run time.

RCV1 dataset. We first assess five systems with a relatively small dataset. The results are shown in Figure 12(a).

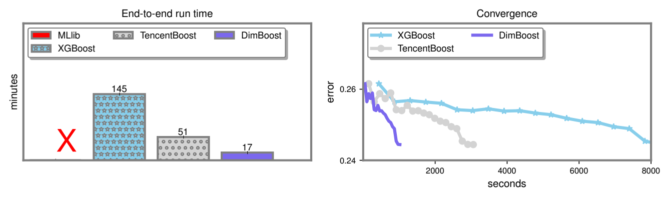
- **Run time.** To begin with, MLlib is much slower than other four systems and it cannot finish the task in a reasonable time. Similar results have been reported in previous works [7]. MLlib cannot work well owing to its inefficient model aggregation and costly histogram construction. XGBoost can finish the training in 17 minutes, while LightGBM is 3× faster. TencentBoost is 1.5× faster than XGBoost, yet slower than LightBM. DimBoost achieves the fastest speed — 4.2× and 1.5× faster than XGBoost and LightGBM. Although RCV1 is a small dataset, DimBoost still beats other systems due to more efficient parameter aggregation and parallel training mechanism.



(a) RCV1 dataset



(b) Synthesis dataset



(c) Gender dataset

Figure 12: End-to-end System Comparison. RCV1 dataset and Synthesis dataset are evaluated in Cluster-1. Gender dataset is evaluated in Cluster-2. We report the end-to-end run time (in minutes) on the left side, and the convergence result on the right side. The convergence curve shows training error in terms of running time (in seconds).

- **Convergence.** All systems achieve comparable accuracy when the training is finished. However, if we assess the training error against time, DimBoost converges the fastest. It is a natural result since DimBoost runs the fastest.

Synthesis dataset. Figure 12(b) presents the experimental results with the Synthesis dataset. Synthesis contains much more instances and features than RCV1.

- **Run time.** In overall, DimBoost needs the least time to train a GBDT model, while MLlib is the slowest. XGBoost finishes in 185 minutes, 9× slower than DimBoost, which demonstrates that XGBoost cannot efficiently handle high dimensional datasets. In contrast, DimBoost only consumes 21 minutes — 3.1× and 5× faster than LightGBM and TencentBoost. By efficiently merging parameters and taking advantage of data sparsity, DimBoost significantly outperforms the other four platforms. The performance improvement of DimBoost verifies that our parameter server solution surpasses the implemented aggregation methods of existing systems for a larger message. Besides, DimBoost gets higher speedup on Synthesis than RCV1. It indicates that DimBoost is more powerful for larger datasets.
- **Convergence.** As can been observed in Figure 12(b), the convergence rate of DimBoost is significantly faster than other systems. It is able to converge to a tolerance within much less time.

Metric	Number of Parameter Servers		
	5	20	50
End-to-end run time (minutes)	38	23	17

Table 4: Impact of Parameter Servers.

Metric	Datasets		
	Gender-10K	Gender-100K	Gender-330K
Test error	0.3014	0.2714	0.2514

Table 5: Impact of Feature Dimension.

7.3.2 *Results over a Large Cluster.*

We proceed to assess the performance of these systems in a real-world productive cluster[1]. As aforementioned, LightGBM is excluded since it cannot support resource management tools (Yarn) and distributed file systems (HDFS). Therefore, LightGBM is not a good choice in an industrial environment.

Gender dataset. We train the Gender dataset, a real dataset, on Cluster-2. The results are presented in Figure 12(c).

- **Run time.** Unfortunately, MLlib fails to produce the results in an endurable time. Similar to the results over Cluster-1, DimBoost can finish in 17 minutes — 8.5× and 3× faster than XGBoost and TencentBoost, respectively. This performance improvement is larger than what we gain on the RCV1 dataset, demonstrating that DimBoost is more suitable for a high dimensional scenario.
- **Convergence.** In accordance with the run time statistics, DimBoost achieves the fastest convergence rate, followed by TencentBoost. XGBoost, however, converges very slow for this large and high dimensional dataset.

7.3.3 *Impact of Parameter Servers.*

To study whether the number of parameter servers influences the performance, we change the number of parameter servers and present the results in Table 4. The experimental settings are the same as Section 7.3.2. The results show that the system deteriorates 2.2× as p decreases from 50 to 5. DimBoost reveals a good scalable ability that it can achieve faster processing speed by adding more parameter servers.

7.3.4 *Impact of Feature Dimensions.*

We next assess the impact of feature dimension. To this end, we use different feature subsets and train them individually using Cluster-2. For instance, *Gender-10K* refers to the dataset that contains the first 10K features of the entire feature set. As shown in Table 5, the use of more features brings higher prediction accuracy. Specifically, using the entire feature set achieves a 20% higher accuracy than using only 10K features. This performance evolution, unsurprisingly, benefits from understanding more information of users.

7.3.5 *Impact of Dimension Reduction.*

As the readers might suspect, does it make a difference if we lower the dimension of the training dataset through dimensionality reduction? To study the impact of dimension reduction method, we use Spark MLlib to perform

[1]We use fifty machines in Cluster-2 since the total storage of training data and gradient histogram is about 9GB. It is almost the same as the maximal memory constraint.

Method	PCA time	Training time	Test error
With PCA	64	9	0.2785
Without PCA	0	17	0.2514

Table 6: Impact of Dimension Reduction. Time is in minutes.

a PCA (Principle Component Analysis) operation on Gender dataset and reduce the dimension to 10K. The results are presented in Table 6. As the results illustrate, dimension reduction is very expensive for a large-scale and high-dimensional dataset. Unfortunately, the overall time cost increases after we apply the PCA operation. Worse still, the reduction of dimensions, as with other feature selection methods, inevitably decreases the buried information inside the data, and therefore deteriorates the model accuracy [13].

7.4 More experiments

Due to the space constraint, we present more experiments in Appendix A.2 and Appendix A.3.

Scalability. We evaluate the scalability in Appendix A.2, including an execution on a single machine. To show the merit and overhead of distribution, we decompose the run time into data loading, computation, and communication.

Low-dimensional dataset. In Appendix A.3, we evaluate the chosen systems on a low-dimensional dataset.

8 RELATED WORK

Gradient boosting decision tree (GBDT) is a powerful machine learning algorithm that can be used to perform a range of tasks, such as classification, regression, and ranking. GBDT trains a few regression trees successively, puts each instance onto a leaf, and gives a continuous prediction [15]. After all the trees are finished, we sum the predictions of all the trees [14]. To train a regression tree, a widely adopted approach is to use a loss function to measure the quality of the tree, build the gradient histograms, and find the optimal split of each tree node. In addition to the first-order gradient, the second-order gradient is useful to enhance the performance [14].

A series of existing works have built GBDT systems, such as R GBM, scikit-learn, MLlib [29], XGBoost [7], LightGBM [28], and TencentBoost [22]. R GBM and scikit-learn are executed on a single machine system, therefore they are not scalable enough to handle a large-scale dataset. MLlib implements GBDT in a distributed environment on top of Spark. As the most prevalent system recently, XGBoost achieves an efficient performance through executing GBDT in parallel. LightGBM is a recently developed system which supports both data-parallel and feature-parallel paradigms. Although performing well in a productive cluster, MLlib and XGBoost might encounter a performance deterioration dealing with a high dimensional dataset. MLlib works under the map-reduce abstraction, and XGBoost chooses all-reduce MPI to conduct the model aggregation. Map-reduce needs a single coordinator to merge local gradient histograms and thereby encounters a single-point bottleneck facing high dimensional features. All-reduce entails several non-overlapping communication steps. In contrast, LightGBM needs a less communication cost than MLlib and XGBoost by using the reduce-scatter approach. However, reduce-scatter still takes multiple communication steps and experiences a performance decline if the number of workers is not a power of two. Worse, the deployment of LightGBM is manual and unreasonable. It cannot satisfy a real-world case where resource management tools (Yarn and Kubernetes) and distributed file systems (HDFS) are common. In our previous work, we developed a system, called TencentBoost [22], to train GBDT in a parameter server architecture. The parameter server architecture [10, 25], which partitions a model parameter over machines, is highly efficient to train high dimensional datasets. TencentBoost simply applies the parameter server architecture to GBDT and lacks a deep investigation into data sparsity and various data representations. We treated TencentBoost as one of the competitors and showed that DimBoost can be up to 5× faster.

A series of relevant works have investigated sparse representations to train ML algorithms faster. For example, Rendle et.al [31] train factorization machine by only accessing nonzero values. However, most of them focus on generalized linear models. Their sparse optimizations cannot be applied in GBDT, a nonlinear and tree based model. Another related work, GBDT-SPARSE, trains GBDT when the output space is high dimensional and sparse, e.g., extreme multilabel classification. Their goal is to reduce model size and prediction time, which is orthogonal to our purpose [33]. The database community also studies techniques to resolve the sparsity problem. Olteanu et. al. propose an in-database factorized learning approach to learn combinations of original features from table joins [32]. This approach can avoid Cartesian product in linear models. Nevertheless, this kind of techniques aim at feature engineering, which are orthogonal to our target.

9 CONCLUSION

We implemented a scalable gradient boosting decision tree system, namely DimBoost. To address the inefficient model aggregation of existing systems, we proposed to use the parameter server architecture. We first introduced a distributed execution plan, and described our parameter management. To build a gradient histogram faster, we proposed a novel sparsity-aware algorithm that utilizes data sparsity to reduce the access of features. We designed an index structure that can be used to access fewer instances. Furthermore, we proposed a set of methods to efficiently find the split of tree nodes, including a task scheduler, a two-phase split strategy, and a low-precision method that compresses histograms. Empirical results on large-scale datasets and real applications demonstrated that DimBoost outperforms the state-of-the-art systems in the presence of high dimensional datasets.

ACKNOWLEDGEMENTS

This research is funded by National Natural Science Foundation of China (No. 61572039, 61702015, U1536201), 973 program (No. 2014CB340405), China Postdoctoral Science Foundation (2017M610020), and PKU-Tencent joint research Lab.

REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. 2016. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467* (2016).

[2] Sanjay Agrawal, Vivek Narasayya, and Beverly Yang. 2004. Integrating vertical and horizontal partitioning into automated physical database design. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data.* 359–370.

[3] Dan Alistarh, Jerry Li, Ryota Tomioka, and Milan Vojnovic. 2016. QSGD: Randomized Quantization for Communication-Optimal Stochastic Gradient Descent. *arXiv preprint arXiv:1610.02132* (2016).
[4] Léon Bottou. 2010. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*. 177–186.
[5] Léon Bottou. 2012. Stochastic gradient descent tricks. In *Neural Networks: Tricks of the Trade*. 421–436.
[6] Leo Breiman. 2001. Random forests. *Machine learning* 45, 1 (2001), 5–32.
[7] Tianqi Chen and Carlos Guestrin. 2016. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*. 785–794.
[8] Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. 2014. Project adam: Building an efficient and scalable deep learning training system. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 571–582.
[9] Carlo Curino, Evan Jones, Yang Zhang, and Sam Madden. 2010. Schism: a workload-driven approach to database replication and partitioning. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 48–57.
[10] Wei Dai, Jinliang Wei, Jin Kyu Kim, Seunghak Lee, Junming Yin, Qirong Ho, and Eric P Xing. 2013. Petuum: A Framework for Iterative-Convergent Distributed ML. *arXiv preprint arXiv:1312.7651* (2013).
[11] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. 2012. Large scale distributed deep networks. In *Advances in neural information processing systems*. 1223–1231.
[12] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.
[13] Imola K Fodor. 2002. *A survey of dimension reduction techniques*. Technical Report. Lawrence Livermore National Lab., CA (US).
[14] Jerome Friedman et al. 2000. Additive logistic regression: a statistical view of boosting. *Annals of statistics* 28, 2 (2000), 337–407.
[15] Jerome H Friedman. 2001. Greedy function approximation: a gradient boosting machine. *Annals of statistics* (2001), 1189–1232.
[16] Shahram Ghandeharizadeh and David J DeWitt. 1990. Hybrid-range partitioning strategy: A new declustering strategy for multiprocessor database machines. In *Proc. 16th international Conference on VLDB*. 481–492.
[17] Amol Ghoting, Rajasekar Krishnamurthy, Edwin Pednault, Berthold Reinwald, Vikas Sindhwani, Shirish Tatikonda, Yuanyuan Tian, and Shivakumar Vaithyanathan. 2011. SystemML: Declarative machine learning on MapReduce. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*. 231–242.
[18] Michael Greenwald and Sanjeev Khanna. 2001. Space-efficient online computation of quantile summaries. In *ACM SIGMOD Record*, Vol. 30. 58–66.
[19] JBoss. 2004. Netty. (2004). http://netty.io/
[20] Jiawei Jiang, Bin Cui, Ce Zhang, and Lele Yu. 2017. Heterogeneity-aware distributed parameter servers. In *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 463–478.
[21] Jiawei Jiang, Ming Huang, Jie Jiang, and Bin Cui. 2017. TeslaML: Steering Machine Learning Automatically in Tencent. In *Asia-Pacific Web (APWeb) and Web-Age Information Management (WAIM) Joint Conference on Web and Big Data*. Springer, 313–318.
[22] Jie Jiang, Jiawei Jiang, Bin Cui, and Ce Zhang. 2017. TencentBoost: A Gradient Boosting Tree System with Parameter Server. In *Data Engineering (ICDE), 2017 IEEE 33rd International Conference on*. 281–284.
[23] Jie Jiang, Lele Yu, Jiawei Jiang, Yuhong Liu, and Bin Cui. 2017. Angel: a new large-scale machine learning system. *National Science Review* (2017), nwx018.
[24] David D Lewis, Yiming Yang, Tony G Rose, and Fan Li. 2004. Rcv1: A new benchmark collection for text categorization research. *Journal of machine learning research* 5, Apr (2004), 361–397.
[25] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. 2014. Scaling distributed machine learning with the parameter server. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 583–598.
[26] Mu Li, Ziqi Liu, Alexander J Smola, and Yu-Xiang Wang. 2016. Difacto: Distributed factorization machines. In *Proceedings of the Ninth ACM International Conference on Web Search and Data Mining*. 377–386.
[27] Xupeng Li, Bin Cui, Yiru Chen, Wentao Wu, and Ce Zhang. 2017. Mlog: Towards declarative in-database machine learning. *Proceedings of the VLDB Endowment* 10, 12 (2017), 1933–1936.
[28] Qi Meng, Guolin Ke, Taifeng Wang, Wei Chen, Qiwei Ye, Zhi-Ming Ma, and Tieyan Liu. 2016. A communication-efficient parallel algorithm for decision tree. In *Advances in Neural Information Processing Systems*. 1279–1287.
[29] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, et al. 2016. Mllib: Machine learning in apache spark. *The Journal of Machine Learning Research* 17, 1 (2016), 1235–1241.
[30] Jun Rao, Chun Zhang, Nimrod Megiddo, and Guy Lohman. 2002. Automating physical database design in a parallel database. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*. 558–569.
[31] Steffen Rendle. 2013. Scaling factorization machines to relational data. In *Proceedings of the VLDB Endowment*, Vol. 6. VLDB Endowment, 337–348.
[32] Maximilian Schleich, Dan Olteanu, and Radu Ciucanu. 2016. Learning linear regression models over factorized joins. In *Proceedings of the 2016 International Conference on Management of Data*. ACM, 3–18.
[33] Si Si, Huan Zhang, S Sathiya Keerthi, Dhruv Mahajan, Inderjit S Dhillon, and Cho-Jui Hsieh. 2017. Gradient Boosted Decision Trees for High Dimensional Sparse Output. In *International Conference on Machine Learning*. 3182–3190.
[34] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. 2005. Optimization of collective communication operations in MPICH. *The International Journal of High Performance Computing Applications* 19, 1 (2005), 49–66.
[35] Stephen Tyree, Kilian Q Weinberger, Kunal Agrawal, and Jennifer Paykin. 2011. Parallel boosted regression trees for web search ranking. In *Proceedings of the 20th international conference on World wide web*. 387–396.
[36] Jinliang Wei, Wei Dai, Aurick Qiao, Qirong Ho, Henggang Cui, Gregory R Ganger, Phillip B Gibbons, Garth A Gibson, and Eric P Xing. 2015. Managed communication and consistency for fast data-parallel iterative analytics. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*. 381–394.
[37] Yahoo. 2004. Data Sketches. (2004). https://datasketches.github.io/
[38] Lele Yut, Ce Zhang, Yingxia Shao, and Bin Cui. 2017. LDA*: a robust and large-scale topic modeling system. *Proceedings of the VLDB Endowment* 10, 11 (2017), 1406–1417.
[39] Hantian Zhang, Kaan Kara, Jerry Li, Dan Alistarh, Ji Liu, and Ce Zhang. 2016. ZipML: An End-to-end Bitwise Framework for Dense Generalized Linear Models. *arXiv:1611.05402* (2016).
[40] Martin Zinkevich, Markus Weimer, Lihong Li, and Alex J Smola. 2010. Parallelized stochastic gradient descent. In *Advances in neural information processing systems*. 2595–2603.

A APPENDIX

A.1 Proof of Low-precision Gradient Histogram

The introduce of low-precision histograms inevitably incurs quantification errors, which might weaken the quality of split results found. In the following part of this section, we will theoretically analyze whether this low-precision method impairs the correctness of the split results.

For each item G_j in a histogram, the compressor introduces a quantification error $e \in (-\frac{|c|}{2^d}, \frac{|c|}{2^d})$ after converting it into G'_j. We make a reasonable assumption that e conforms to a uniform distribution, that is, $e \sim U(-\frac{|c|}{2^d}, \frac{|c|}{2^d})$. As described in Algorithm 1, the left sum of histogram buckets is calculated as follows:

$$G_L = \sum_L G_j, H_L = \sum_L H_j$$

Therefore, the compressed histogram bucket can be calculated as:

$$G'_L = \sum_L G'_j = \sum_L (G_j + e_j) = G_L + \sum_L e_j$$

Since e_j conforms a uniform distribution, the expectation of G_L remains the same after compressing since $E[\sum e_j]$ equals zero. For the same reason, the expectation of H_L stands the same with the low-precision approach. With the compressed histogram buckets, the split entry we find is the one that contributes the maximal gain:

$$j = \arg\max_j \frac{G'^2_L}{H'_L + \lambda} + \frac{(G - G'_L)^2}{(H - H'_L) + \lambda} - \frac{G^2}{H + \lambda}$$

Being able to maintain the equality of the expectation of histogram buckets, we assure that the maximal objective gain we obtain has the same expectation as the original gain.

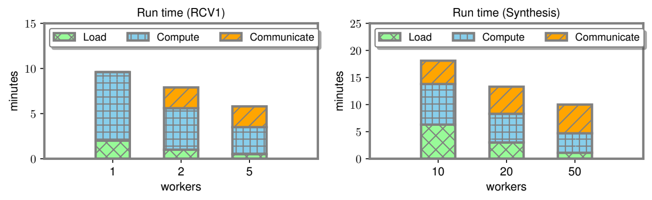


Figure 13: Scalability. RCV1 and Synthesis are evaluated on Cluster-2. We report the run time (in minutes) and decompose the run time into loading, computation, and communication.

A.2 Scalability

To assess the scalability of our method, we change the number of machines and train the datasets on Cluster-2. Since the variation of machines will influence the time to load data from HDFS, we break down the time overhead into data loading, computation, and communication. The other settings are the same as Section 7.3.1.

RCV1. RCV1 can be run on a single machine since it is a relatively small dataset. With a single machine, the communication cost is avoided. We then increase the number of workers to two. Although using two machines brings communication cost, the overall run time is reduced by 18%. According to the breakdown of run time, the reason is obvious. On the one hand, the time cost of data loading decreases proportionally with the increase of machines. On the other hand, the computation time decreases by 39.5% owing to the benefit of data parallelism. The use of five machines further lowers the time of data loading and computation — 49% and 35% respectively.

Synthesis. For a larger dataset Synthesis, it is infeasible to use a single machine since Cluster-2 issues a 10GB memory restriction on a single machine for each task. We initially use 10 machine, and then increase the number to 20 and 50. As can be seen, the time of data loading decreases linearly. The time cost of computation achieves a sublinear speedup — 2.1× faster when the number of workers increases 5×. A linear speedup is not achieved because a few calculation procedures of GBDT have no relations to the number of training instances, such as the phase of finding split. Generally, the increase of machines incurs higher communication overhead. DimBoost, however, benefits from the parameter server architecture which empowers efficient communication. The communication cost does not significantly increase when we use more workers.

Summary. The above results indicate that the distribution of GBDT is able to accelerate the training of GBDT. Data loading and computation consume significantly less time given more machines. Meanwhile, the introduce of more machines does not obviously incur higher communication overhead. Overall, DimBoost achieves a higher speedup on a larger dataset than on a smaller dataset, demonstrating that DimBoost reveals merits in a large-scale setting.

A.3 Experiments on Low Dimensional Dataset

In the main body, we benchmark high dimensional datasets. It raises a question how our system performs on a low dimensional dataset. In this section, we use a low-dimensional synthesis dataset Synthesis-2 which consists of 100 million instances and 1000 features. The other settings are the same as Section 7.3.2.

As Figure 14 shows, DimBoost still achieves the best performance compared with other competitors. It is 7.8× and 4.5× faster than XGBoost and TencentBoost, respectively. For a low dimensional dataset, the communication cost is significantly smaller than the computation cost. This performance improvement mainly derives from our delicately designed paradigm for parallel training.

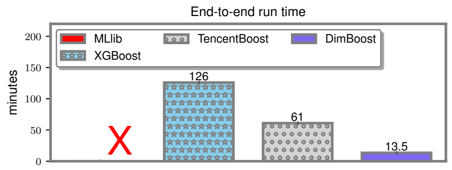


Figure 14: Results on a low dimensional dataset.