TencentBoost: A Gradient Boosting Tree System with Parameter Server

Jie Jiang[†§] Jiawei Jiang[‡§] Bin Cui[♯] Ce Zhang[♮]
[†]School of Software & Microelectronics, Peking University [§]Tencent Inc.
[‡]School of EECS & Key Laboratory of High Confidence Software Technologies (MOE), Peking University
[♯] The Shenzhen Key Lab for Cloud Computing Technology & Applications, School of ECS, Peking University
[♮] Department of Computer Science, ETH Zürich
Email: [†]1401110619@pku.edu.cn [‡]blue.jwjiang@pku.edu.cn [♯]bin.cui@pku.edu.cn [♮]ce.zhang@inf.ethz.ch

***Abstract*—Gradient boosting tree (GBT), a widely used machine learning algorithm, achieves state-of-the-art performance in academia, industry, and data analytics competitions. Although existing scalable systems which implement GBT, such as XGBoost and MLlib, perform well for datasets with medium-dimensional features, they can suffer performance degradation for many industrial applications where the trained datasets contain high-dimensional features. The performance degradation derives from their inefficient mechanisms for model aggregation — either map-reduce or all-reduce. To address this high-dimensional problem, we propose a scalable execution plan using the parameter server architecture to facilitate the model aggregation. Further, we introduce a work partition strategy, a sparse-pull method, and an efficient index structure to increase the processing speed. We implement a GBT system, namely TencentBoost, in the production cluster of Tencent Inc. The empirical results show that our system is 2-20$\times$ faster than existing platforms.**

I. INTRODUCTION

Machine learning (ML) techniques have shown their spectacular capabilities to mine valuable insights from large volume of data. There are two major concerns in large-scale ML — effective models to learn knowledge from raw data, and distributed ML systems to deal with the challenge of big data which exceeds the processing capability of one machine.

Among various ML methods, boosting strategy has been proved to be successful via building weak learners to successively refine performance. Particularly, gradient boosting tree (GBT), which uses tree as the base weak learner, is prevailing recently, both in academia and industry [1]. It is also one of the most preferred choices in data analytics competitions such as Kaggle and KDDCup. Through using the boosting strategy to build multiple trees, the popularity of GBT comes from its state-of-the-art performance on many ML workloads, ranging from classification, regression, feature selection to ranking. Many platforms have provided plans to execute GBT, such as scikit-learn [1], R GBM [2], pGBRT [2], MLlib [3], and XGBoost [4]. Among them, MLlib and XGBoost are two production-level distributed systems.

Since the explosive increase of data volume overwhelms the capability of single machine, it is inevitable to deploy GBT in a distributed environment. Distributed implementations of GBT generally follow the following procedures.

1) The training instances are partitioned onto a set of workers.
2) To split one tree node, each worker computes the gradient statistics of the instances. For each feature, an individual gradient histogram needs to be built.
3) A coordinator aggregates the gradient histograms of all workers, and finds the best split feature and split value.
4) The coordinator broadcasts the split result. Each worker splits the current tree node, and proceeds to new tree nodes.

Since more features generally yield higher predictive accuracy in practice, many datasets used in industrial applications often contain hundreds of thousands or even millions of features. Considering that GBT requires merging of gradient histograms for all the features during each iteration, when the dimension of features increases, the communication cost of model aggregation proportionally increases meanwhile. However, existing systems fail to handle this high-dimensional scenario efficiently owing to their inefficient model aggregation. For example, MLlib employs map-reduce [5], [6] framework to merge parameters during training, while XGBoost chooses all-reduce MPI to summarize local solutions. For parameters with considerably large size, both map-reduce and all-reduce encounter a single-point bottleneck. Therefore, their paradigms of model aggregation are ill-suited for high-dimensional features. An alternative to map-reduce and all-reduce is the parameter server architecture, which partitions the parameters across serveral machines to avoid the single-point bottleneck. Prevalent parameter server systems, such as Petuum [7] and TensorFlow [8], provide efficient distributed implementations for linear algorithms (LR, SVM), neural network algorithms (CNN), and clustering algorithms (KMeans). These ML algorithms usually follow the training pattern that each worker pulls the model parameter, reads training instances, and pushes updates to the parameter server. Unfortunately, they do not provide solutions for GBT as the distributed training process of GBT is more complex than the aforementioned ML algorithms.

To address the high-dimensional challenge, we propose an execution plan with the parameter server architecture. To accelerate the process of split-find, we schedule the workers via a round-robin partition strategy to let them cooperatively conduct the split-find work of different tree nodes. As a further improvement, we introduce a sparse-pull mechanism in which the parameter servers conduct the split-find work and return split results to the workers. Besides, we design an index structure to speed up the parallel execution of building gradient

[1]http://scikit-learn.org/stable/
[2]https://cran.r-project.org/web/packages/gbm/index.html

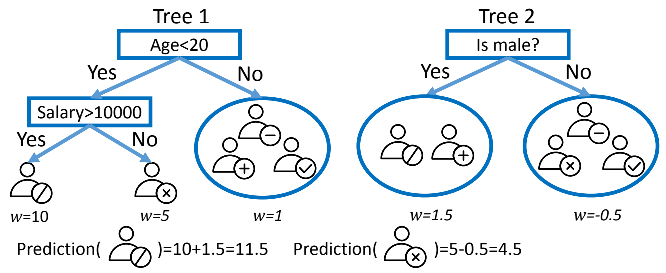

Fig. 1: An example of gradient boosting tree

histogram. This structure prevents scanning the whole dataset by mapping the instances to the tree nodes. The major technical contributions of this paper can be summarized as follows:

- We propose a distributed execution plan for GBT. Through distributing the cost of model aggregation over parameter servers, we can efficiently support high-dimensional features.
- We design two network optimizations to facilitate the split-find operation — a round-robin strategy to schedule the workers and a sparse-pull mechanism for the parameter servers.
- We engineer an efficient index structure to accelerate the parallel execution of building gradient histograms.
- The empirical results show that our implemented system TencentBoost can be 2-20× faster than the state-of-the-arts.

The rest of this paper is organized as follows. We present the related work and preliminaries in Section II. We describe the proposed distributed execution plan in Section III. The network optimization approaches are presented in Section IV. We describe the details of parallel training in Section V and present experiments in Section VI. We conclude in Section VII.

II. Related Work and Preliminaries

In this section, we introduce the related work and preliminaries related to gradient boosting tree and the parameter server architecture that our implemented system employs.

A. Data Model

Previous works have studied different data layouts for storing both types of data as matrices and vectors [9]. One popular data model, which we also adopt, decouples the data used during training into immutable data and mutable data. The training features and labels are immutable, while the predictive model that we want to find will be updated frequently.

We formalize the data model used in this work as follows. The input dataset contains training instances and their labels: $\{\mathbf{x}_i, y_i\}_{i=1}^{N}$, where N is the number of training instances, $\mathbf{x}_i \in \mathbb{R}^n$ is the feature vector, and $y_i \in \mathbb{R}$ is the label.

B. Gradient Boosting Tree

Additive prediction pattern. Gradient boosting tree (GBT) [3] belongs to the tree ensemble model [10]. Figure 1 gives an example of GBT. In one tree f_t, each training instance $\mathbf{x}_i$ is classified to one leaf. Each leaf predicts its instances with a leaf weight w. After building one tree, GBT updates the prediction of each instance by adding the corresponding leaf weight. Then GBT moves to the next tree. GBT adopts the regression tree that gives continuous predictive weight on one leaf, rather than the decision tree in which each leaf gives predictive class. Once finishing all the trees, GBT sums the predictions of all the trees as the final prediction [11].

[3]a.k.a. gradient boosting machine or gradient boosting regression tree

Algorithm 1 Greedy Split-find Algorithm

```
$M$: # features, $N$: # instances, $K$: # split candidates
1: for $m = 1$ to $M$ do
2:     generate $K$ split candidates $S_m = \{s_{m1}, s_{m2}, ..., s_{mk}\}$
3: end for
4: for $m = 1$ to $M$ do
5:     loop $N$ instances to generate gradient histogram with $K$ bins
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
```

$$\hat{y}_i = \sum_{t=1}^{T} f_t(\mathbf{x}_i) \tag{1}$$

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

Training method. Unlike linear and neural models which can be trained with gradient-based optimization method, GBT is trained in an additive manner. For the t-th tree, we need to minimize the following regularized objective.

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

Here l is a loss function that calculates the loss given the prediction and the target, such as the logistic loss and the square loss. Ω denotes the regularized penalty function to avoid over-fitting. We follow the choice of XGBoost [4] which has been proved to be effective:

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

where L denotes the number of leaves in the tree, and w denotes the weights of leaves. γ and λ are hyper-parameters.

LogitBoost [11] expands $F^{(t)}$ via second-order approximation. g_i and h_i are the first and second order gradients of l.

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

Using I_j=$\{i|p(\mathbf{x}_i)$=$j\}$ to denote the instances belonging to the j-th leaf and removing the constant term, we can calculate the approximation of $F^{(t)}$ as:

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

If all the instances have been placed in the tree, the optimal weight value of the j-th leaf and the optimal objective are:

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

We can enumerate every possible tree structure to find the optimal solution. However, this scheme is arguably impractical in practice. Therefore, we generally adopt a greedy algorithm to successively split the tree nodes, as illustrated in Algorithm 1. Given K split candidates for each feature, we loop all the instances of the node to build a gradient histogram that summaries the gradient statistics (line 4-8). Specially, G_{mk} sums g_i of the instances whose m-th features fall into the span of the $(k$–$1)$-th and the k-th split candidates, and H_{mk} sums h_i in the same manner. After building the gradient histogram which includes the gradient statistics of all the features, we enumerate all the histogram bins and find the split that gives the maximal objective gain (line 10-17).

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

where I_L and I_R are the left and the right children nodes after splitting. Note that we can propose split candidates with several methods. The exact proposal sorts all the instances by each feature and uses all possible splits. However, in the era of large-scale dataset and distributed ML, iteratively sorting is too time-consuming. Consequently, a widely employed strategy is to propose limited split candidates according to the percentiles of feature distribution. Many works have designed quantile sketch algorithms to provide distributed solutions for this purpose, such as GK [12], DataSketches [13], and WQS [4].

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 which has been proved to be effective in practice [4].

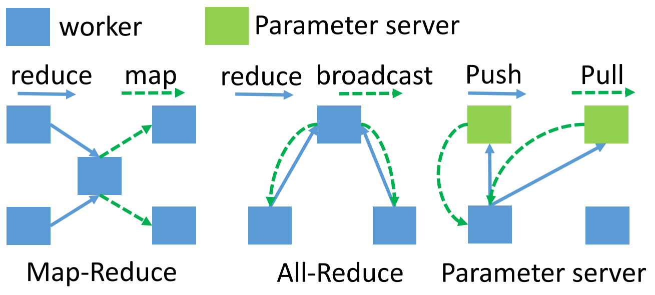

Fig. 2: Model aggregation approaches

C. *Gradient boosting tree systems*

Some existing works have built GBT systems, such as R GBM, scikit-learn, pGBRT [2], LightGBM [14], MLlib [3], and XGBoost [4]. R GBM and scikit-learn are executed on single machine, while MLlib and XGBoost are distributed platforms. Some systems support specific task, e.g., pGBRT is proposed to address ranking task. Among them, MLlib and XGBoost are two most widely used platforms for productive applications. MLlib works under the map-reduce abstraction, and XGBoost chooses all-reduce MPI to conduct the model aggregation. As shown in Figure 2, both map-reduce and all-reduce need a single coordinator to merge local gradient histograms and broadcast split results. Unsurprisingly, they encounter a single-point bottleneck facing high-dimensional features because the size histogram is proportional to the number of features.

D. *Parameter Server*

The parameter server (PS) architecture in Figure 2 is a promising candidate to address the challenge of aggregating high-dimensional parameter [7]. Several machines together store a parameter to prevent the single-point bottleneck, and provide interfaces for the workers to push and pull parameters. Each worker holds a local copy of the parameter, and periodically pushes parameter updates to the PS. To address the high-dimensional challenge of GBT, the PS framework is a promising candidate.

III. Distributed Execution Plan

In this section, we first describe the execution plan on each worker, then elaborate the parameter management on the PS.

A. *Worker*

We decompose the training procedure on each worker into seven phases, as illustrated in Figure 3. Among all workers, a leader worker is designated to perform some specific work that only one worker is allowed.

1) ***Create sketch:*** Each worker uses its assigned data shard to generate local quantile sketches (one sketch for one feature). Then, these local quantile sketches are pushed to the PS.
2) ***Pull sketch:*** Each worker pulls the merged sketches from the PS and proposes split candidates for each feature.
3) ***New tree:*** Each worker creates a new tree, performs some initial work, i.e., initializing the tree structure, computing the

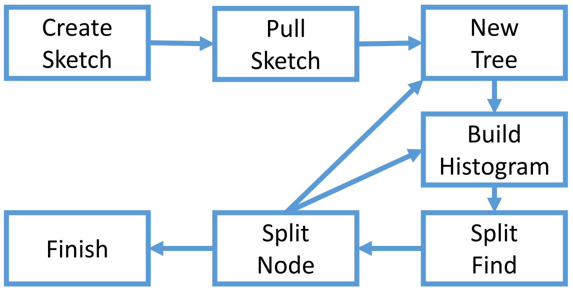

Fig. 3: Training procedure of worker

Algorithm 2 The execution plan of worker

```
1: $phase = CREATE\_SKETCH$
2: while $phase\ != FINISH$ do
3:   if $phase == CREATE\_SKETCH$ then
4:     $create\_sketch()$; $phase = PULL\_SKETCH$; continue;
5:   else if $phase == PULL\_SKETCH$ then
6:     $pull\_sketch()$; $phase = NEW\_TREE$; continue;
7:   else if $phase == NEW\_TREE$ then
8:     $new\_tree()$; $phase = BUILD\_HISTOGRAM$; continue;
9:   else if $phase == BUILD\_HISTOGRAM$ then
10:    $build\_histogram()$; $phase = SPLIT\_FIND$; continue;
11:  else if $phase == SPLIT\_FIND$ then
12:    if is leader worker then
13:      $split\_find()$;
14:    end if
15:    $phase = SPLIT\_NODE$; continue;
16:  else if $phase == SPLIT\_NODE$ then
17:    $split\_node()$;
18:    if has active nodes then
19:      $phase = BUILD\_HISTOGRAM$;
20:    else if has next tree then
21:      $phase = NEW\_TREE$;
22:    else
23:      $phase = FINISH$;
24:    end if
25:  end if
26: end while
27: if is leader worker then
28:   $finish()$;
29: end if
```

gradients, and setting the root node to active. The leader worker samples a subset of features and pushes it to the PS.

4) ***Build histogram:*** If the active node is the root of the tree, each worker pulls the sampled features from the PS. Each worker uses its assigned data to build gradient histograms for the active tree nodes, and pushes them to the PS.

5) ***Split find:*** The leader worker pulls the merged histograms of active nodes from the PS and finds the best split. Then the leader worker pushes the split results to the PS.

6) ***Split node:*** Each worker (1) pulls the split results from the PS, (2) splits the active tree nodes, (3) adds children nodes to the tree, and (4) deactivates the active nodes. If the depth of the tree is less than the maximal depth, the worker sets the tree nodes in the next layer to active.

7) ***Finish:*** The leader worker outputs the GBT model.

To ensure that different workers proceed in the same pace, we introduce an iterative plan that one worker cannot start the next iteration until all workers have finished the current iteration. The detailed execution plan is elaborated in Algorithm 2. Each worker checks its current phase to determine what to do in each iteration. The operations of $create\_sketch()$ and $pull\_sketch()$ are called only once by each worker (line 3-6), while the operations of $build\_histogram()$, $split\_find()$, and $split\_node()$ are iteratively executed (line 9-17). At the end of the *SPLIT_NODE* phase, if there exists an active node to split, the worker switches to the *BUILD_HISTOGRAM* phase (line 19). If we finish building the current tree, the worker updates the predictions and proceeds to the next tree (line 21). Once finishing building all the trees, the worker stops training. We employ a layer-wise scheme to consecutively add active nodes [2]. In other words, after splitting the current layer, we set the tree nodes of the next layer to active and continue the next split round.

B. Parameter Server

Parameter storage. Parameters are represented as vectors in our system. We use several vectors to represent matrix data. For dense data, we store the values of each dimension. For sparse data, we store the ordered indexes and the corresponding values of non-zero entries to reduce the memory cost.

Parameter partition. Adopting the PS architecture, it is a prerequisite to design how to partition a parameter over several machines. The problem of data partitioning across a set of nodes has been extensively studied by the database community [15], for both single system [16] and shared-nothing systems [17]. The most widely used approaches are round-robin partition (successively partition), range partition (partition based on the values of the key), and hash partition (partition with a hash function). Hash partition achieves more balanced query distribution, while range partition facilitates range queries [17]. In the context of parameter server architecture, the queries happen in diverse patterns. Some tasks need to get the whole parameter, while others need a portion of the parameter. To achieve a trade-off between query balance and fast range query, we adopt a hybrid range-hash strategy [18]. We first partition a parameter to several ranges based on the indexes, and use hash partition to put each partition to one node. Users can set the partition size according to their specific scenario. The default partition size is calculated by dividing the dimension of the parameter by the number of parameter servers.

Parameter manipulation. We provide two user-defined functions for the workers 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.

Parameter layout. The parameter layout of our implementation is illustrated in Figure 4. There are five primary parameters on the PS — the quantile-sketches, the sampled-features, the gradient-histograms, the split-features, and the split-values.

The sketches (one sketch per feature) are concatenated to form the parameter of quantile-sketches. The number of partitions is equal to P (# parameter servers). Since one feature's sketch on the PS is generated by merging the local sketches of all workers, we leverage the user-defined *Push* function to implement the tailored sketch merging function.

The parameter of sampled-features stores the subset of features used in the current tree. The parameter of split-features

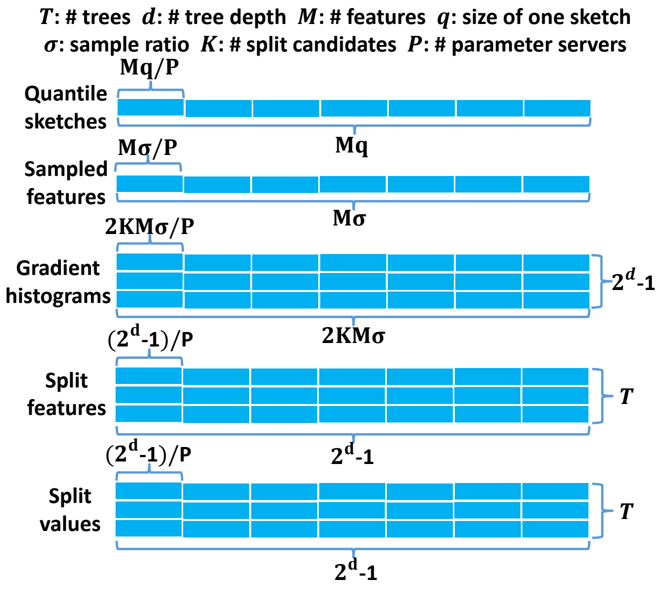

Fig. 4: Parameter layout

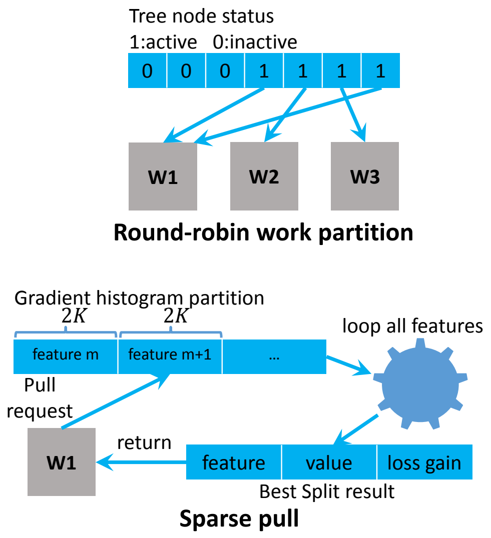

Fig. 5: Network optimization

and split-values store the split results of all the trees. These three parameters use the default partition size. The parameter of gradient-histograms contains 2^d–1 rows (# maximal tree nodes) where d denotes the maximal tree depth. Each row stores the gradient histogram of sampled features for one node — $\{G_{11},...,G_{1K},H_{11},...,H_{1K},G_{21},...\}$. We divide one row into P partitions while assuring that one feature's gradient summary is in one partition.

IV. Network Optimizations

Despite of the proposed execution plan that facilitates the aggregation of gradient histogram, there is still one time-consuming phase, i.e., the *SPLIT_FIND* phase. Since each gradient histogram is partitioned over several machines, we need to design a mechanism to find the best split from these distributed partitions. In Algorithm 2, the leader worker pulls the gradient histogram and performs the split-find operation. However, as Figure 4 shows, the gradient histogram can be very large with high-dimensional features. As the tree goes deeper, the number of the active tree nodes increases as well — at most 2^{d-1}. And the communication cost is proportional to the number of the active nodes. To speed up the split-find operation, we propose two optimizations.

Round-robin work partition. Rather than putting the split-find work on the leader worker, we adopt a round-robin scheme to partition the split-find work of active nodes among the workers, as shown in Figure 5. More explicitly, each worker stores the status of all the tree nodes in an array, where the $(2i+1)$-th item and the $(2i+2)$-th item are the children nodes of the i-th item. Each worker scans this array and finds the responsible active nodes according to the round-robin partition. Then it pulls the gradient histograms of responsible active nodes from the PS and performs split-find.

Sparse pull. Existing platforms generally use the parameter server as a distributed key-value store. Although some works [7] provide user-defined *Push* function to support different update mechanisms, they do not consider the flexibility of *Pull* function to support diverse requirements of parameter queries. Our system, however, is able to customize the *Pull* function. We thereby move part work of split-find from the workers to the PS. In Figure 5, when receiving a pull request of one partition from the workers, we operate as follows.

1) We implement the split-find operation of Algorithm 1 (line 10-17) in the *Pull* function.
2) As described in Section III-B, each partition of a gradient histogram contains the gradient summaries of several features. The user-defined *Pull* function loops the gradient summaries to find the optimal split result of this partition. Then the split feature, split value, and objective gain are summarized to form a sparse vector with six double figures.
3) The PS returns the sparse-format vector to the worker.

For the worker, after finding the responsible active nodes with the round-robin strategy of work partition, it pulls the corresponding gradient histograms from the PS. For one gradient histogram, since each returned partition contains the optimal split result for its storing features, the worker only needs to find the optimal split from all the partitions. With this method, we reduce the communication overhead to a large extent. More detailedly, we compress the transferred size of a gradient histogram to $6P$ — a marginal constant size since P is generally less than 1000 in practice.

Analysis of communication cost. We compare the communication cost of different aggregation schemes in Table I. We consider one round of node split here. *map-reduce* uses one node to collect local histograms, while *all-reduce* uses a binomial tree which takes $2\log W$ steps of transmission [19]. The time cost of *parameter server* is inversely proportional to P, hence we can increase P to accelerate the speed. For example, choosing P=W renders *parameter server* $W\times$ and $\log W\times$ faster than *map-reduce* and *all-reduce*.

V. Local parallel training

During the *BUILD_HISTOGRAM* phase, two tree nodes can be processed in parallel if they have no parent-child relation.

Method	# transmission	communication cost	time cost
map-reduce	2	$W(H+R)$	$\frac{W(H+R)}{B}$
all-reduce	2log W	$(H+R)\log W$	$\frac{(H+R)\log W}{B}$
parameter server	2	$W(H+PR)$	$\frac{W(H+PR)}{PB}$

TABLE I: Comparison of communication cost. W: # workers,P: # parameter server,H: histogram size,R: split result's size,B: bandwidth

We describe our parallel strategy in this section, including an efficient index structure and the parallel execution plan.

Instance Layout and Instance-to-node Index. As different threads can build the gradient histograms of two tree nodes in parallel, they need to read the dataset simultaneously. Scanning the whole dataset can be unnecessarily time-consuming, therefore, we design an index structure mapping the instances to the tree nodes.

Figure 6 illustrates our instance layout and instance-to-node index. The storage format of instance is the same as parameter, as described in Section III-B. Since most of high-dimensional datasets are sparse, Figure 6 chooses the sparse format as an example. We use a list to store all the instances so that we can read one instance with instance index. To begin with, we use an array to store the indexes of all the instances. They belong to the 0-th node, i.e., the root node. We thereby define that the span of the 0-th node is from zero to four. Given the split result of the 0-th node (the first feature < 0.05), we rearrange the array to put the instances of the 1-th node to the left and put those of the 2-th node to the right. We scan the array from two directions and swap the instances with wrong place according to the split result (e.g. swap the 0-th instance and the 4-th instance). Then we store the span of two children nodes. With this index structure, each thread is able to rapidly find corresponding instances, instead of scanning all instances.

Parallel execution. On each worker, we establish a thread pool. Industrial cluster managers normally impose memory limit on each worker process. In order to prevent the out-of-memory failure, we calculate the maximal number of threads in the thread pool according to the memory budget and the size of each histogram. The parallel execution is as follows:

1) Each worker starts several threads for the active tree nodes to build gradient histograms in parallel.
2) Each thread sets the status of the tree node to inactive after building the histogram.
3) The worker process proceeds to the next training phase if all the local threads terminate.

VI. Evaluation

A. Experimental Setup

System implementation. TencentBoost is programmed in Java and deployed on Yarn. We use Netty to manage message passing. We store the training data in HDFS and design a data-splitter module to partition data. We implement GK quantile sketch [12] in our system. TencentBoost has been used in many applications of Tencent Inc. for months [20], [21].

Datasets. As shown in Table II, we choose two datasets. The RCV1 dataset [22] is a medium-size news dataset. *Gender*, a dataset with high-dimensional features, is extracted from the user information of WeChat, one of the most popular instance message tools in the world. It is used to predict user's gender. 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 with the first 10K features.

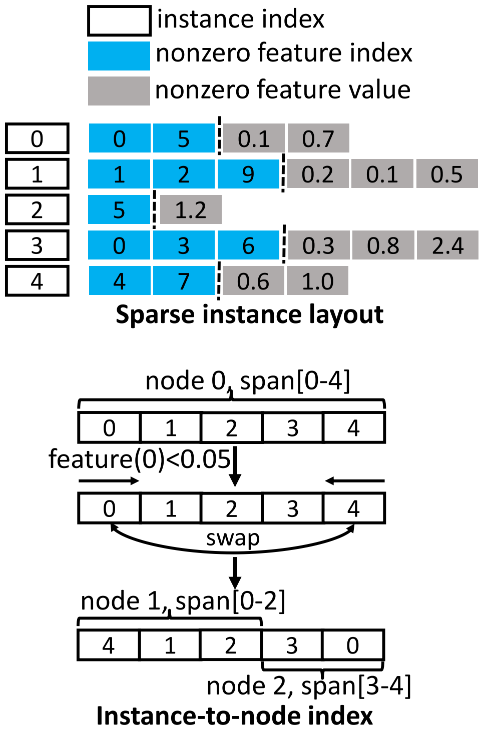

Fig. 6: Instance layout and instance-to-node index

Dataset	# instance	# features	Task
RCV1	700K	47K	News classification
Gender	122M	330K	Gender classification

TABLE II: Datasets for evaluation

Experiment setting. We compare TencentBoost with MLlib and XGBoost, two popular production-level systems. As Tencent and many other industrial companies generally use cluster managers, such as Yarn and Kubernetes, to deploy distributed ML tasks, we do not consider some systems without sufficient distributed support, such as LightGBM. All experiments are conducted on a productive Yarn cluster consisting of 550 machines, each of which is equipped with 64GB RAM, 24 cores, and 1GB Ethernet. We use fifty workers for all systems, and use ten parameter servers for TencentBoost. For hyper-parameters, we choose σ=0.8 (the feature sampling ratio), T=100 (# trees), d=7 (# maximal tree depth), K=100 (# split candidates), and η=0.1 (the shrinkage learning rate). To guarantee the robustness of the results, we take three runs for each experiment and report the average.

B. Experimental Result

Impact of optimizations. We first use our system to investigate the network optimizations and the instance-to-node index proposed in Section IV and Section V. We use *Gender*-330K to evaluate these methods and present the results in Figure 7.

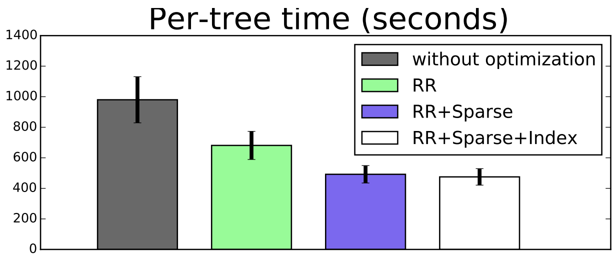

Fig. 7: Impact of optimizations. RR refers to RR work-partition. Sparse refers to sparse-pull. Index refers to instance-to-node index.

Through letting all workers together handle the work of split-find, employing the strategy of round-robin partition is 1.44× faster than the basic version. The sparse pull approach additionally brings 1.38× speedup. In this case, one node's histogram contains 66M figures, but the sparse pull approach only needs to transfer 60 figures. Applying the instance-to-node index further decreases the per-tree time by 17 seconds.

System comparison. We assess the performance of three systems with both datasets. Figure 8 shows the results on the *RCV1* dataset. XGBoost and TencentBoost are 4.5× and 4.96× faster than MLlib because the map-reduce mechanism puts too much pressure on one worker during model aggregation. Then we consider the *Gender* dataset and present the results in Figure 9. With the increase of used features, we observe that the test AUC increases. It demonstrates that more features can improve the prediction accuracy. The per-tree time of XGBoost increases by 11.5× when the number of features increase by 6×, meaning that XGBoost suffers congested model aggregation with larger parameters. For *Gender*-50K, TencentBoost is 1.11× and 4.79× faster than XGBoost and MLlib. While for Gender-330K with more features, TencentBoost achieves larger speedup — 2.55× and 21× respectively. The performance improvements indicate that our system significantly outperforms other two competitors in the presence of high-dimensional features.

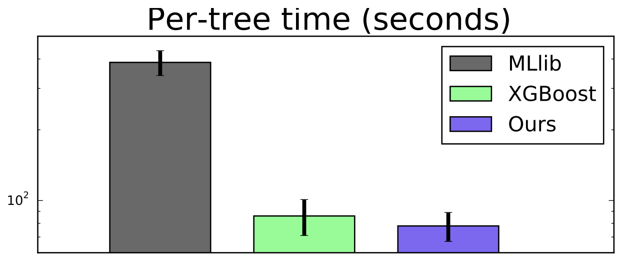

Fig. 8: System comparison with *RCV1* dataset

Impact of parameter server number. We next investigate the impact of P, the number of parameter servers. We vary the value of P while fixing other configurations. As Figure 10 illustrates, choosing smaller P=5 incurs 1.39× slower training, while choosing larger P=20 brings 1.23× speedup. Due to the scalability of the PS framework, we can extend the size of PS to achieve better performance or to support larger parameter.

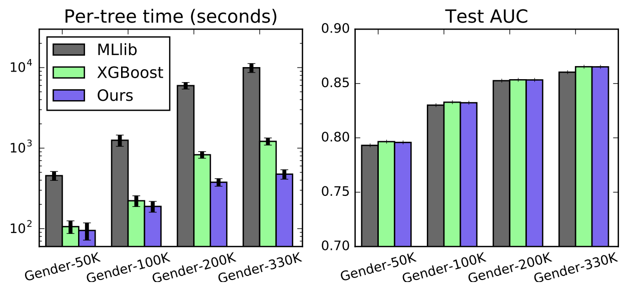

Fig. 9: System comparison with *Gender* dataset

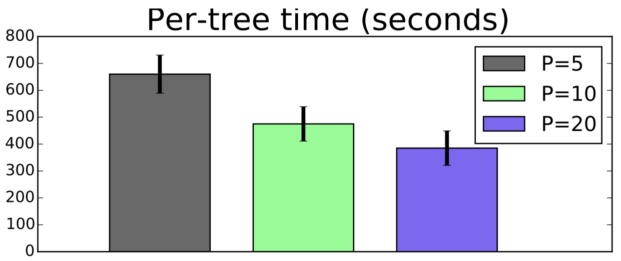

Fig. 10: Impact of parameter server number

VII. Conclusion

We implemented a gradient boosting tree system, TencentBoost, in Tencent Inc. 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. We further designed two efficient network optimizations to reduce the communication cost. Additionally, we engineered an instance-to-node index structure to facilitate the parallel execution of the tree nodes. Empirical results on large-scale datasets and real applications in Tencent Inc. demonstrated that TencentBoost outperforms the state-of-the-art systems in the presence of high-dimensional features.

Acknowledgements

This research is supported by the National Natural Science Foundation of China under Grant No. 61572039, 973 program under No. 2014CB340405, Shenzhen Gov Research Project JCYJ20151014093505032, and Tecent Research Grant (PKU).

References

[1] X. He, J. Pan *et al.*, "Practical lessons from predicting clicks on ads at facebook," in *ADKDD*, 2014, pp. 1–9.

[2] S. Tyree, K. Q. Weinberger, K. Agrawal, and J. Paykin, "Parallel boosted regression trees for web search ranking," in *WWW*, 2011, pp. 387–396.

[3] "Spark mllib," http://spark.apache.org/mllib/.

[4] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *KDD*, 2016, pp. 785–794.

[5] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[6] A. Ghoting, R. Krishnamurthy, E. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian, and S. Vaithyanathan, "Systemml: Declarative machine learning on mapreduce," in *ICDE*, 2011, pp. 231–242.

[7] J. Wei *et al.*, "Managed communication and consistency for fast data-parallel iterative analytics," in *SoCC*, 2015, pp. 381–394.

[8] M. Abadi *et al.*, “Tensorflow: Large-scale machine learning on heterogeneous distributed systems,” *arXiv:1603.04467*, 2016.

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

[10] J. H. Friedman, “Greedy function approximation: a gradient boosting machine,” *Annals of statistics*, pp. 1189–1232, 2001.

[11] J. Friedman *et al.*, “Additive logistic regression: a statistical view of boosting,” *Annals of statistics*, vol. 28, no. 2, pp. 337–407, 2000.

[12] M. Greenwald and S. Khanna, “Space-efficient online computation of quantile summaries,” in *SIGMOD*, vol. 30, no. 2, 2001, pp. 58–66.

[13] “Data sketches,” https://datasketches.github.io/.

[14] “Lightgbm,” https://github.com/Microsoft/LightGBM.

[15] J. Rao, C. Zhang, N. Megiddo, and G. Lohman, “Automating physical database design in a parallel database,” in *SIGMOD*, 2002, pp. 558–569.

[16] S. Agrawal, V. Narasayya, and B. Yang, “Integrating vertical and horizontal partitioning into automated physical database design,” in *SIGMOD*, 2004, pp. 359–370.

[17] C. Curino, E. Jones, Y. Zhang, and S. Madden, “Schism: a workload-driven approach to database replication and partitioning,” *VLDB*, vol. 3, no. 1-2, pp. 48–57, 2010.

[18] S. Ghandeharizadeh and D. J. DeWitt, “Hybrid-range partitioning strategy: A new declustering strategy for multiprocessor database machines,” in *VLDB*, 1990, pp. 481–492.

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

[20] J. Jiang, L. Yu, J. Jiang, Y. Liu, and B. Cui, “Angel: A new largescale machine learning system,” *NSR*, 2017.

[21] J. Jiang, B. Cui, C. Zhang, and L. Yu, “Heterogeneity-aware distributed parameter servers,” in *SIGMOD*, 2017.

[22] D. D. Lewis, Y. Yang, T. G. Rose, and F. Li, “Rcv1: A new benchmark collection for text categorization research,” *JMLR*, vol. 5, no. Apr, pp. 361–397, 2004.