SparkTree: Push the Limit of Tree Ensemble Learning

Hucheng Zhou HUZHO@MICROSOFT.COM
Microsoft Research,
Beijing, China

Bo Zhao BHOPPI@OUTLOOK.COM
Nanjing University,
Nanjing, China

Editor: Kevin Murphy and Bernhard Schölkopf

Abstract

Decision trees, when combined with different loss functions and ensemble techniques, are widely used in web applications such as document ranking in web search and click prediction in ads targeting. Therefore, there is an increasing motivation on an `integrated` framework that supports all loss functions and all ensemble techniques instead of each with a separated system. The web-scale training is challenging because of `big data` that has hundreds of millions of samples and each sample with thousands of features, and `big model` in which thousands of trees are fitted and each tree has hundreds of nodes. Therefore, there is an increasing demand for a `scalable` distributed learning system that exploits all dimensions of parallelism. However, widely used offers such as XGBoost and Spark MLlib are incomplete without the support of ranking such as LambdaMART, and without the support of the feature parallelism, they are not scalable to support a large number of features. Simply adding these supports does not meet the `efficiency` requirement needed to balance the training speed and accuracy. In this paper, we present SparkTree which seamlessly integrates all parallelism with completely new tradeoffs and presents a series of optimizations to improve training speed and accuracy. SparkTree has been evaluated against our production workloads against XGBoost and MLlib, the result indicates that SparkTree outperforms MLlib with a 8.3X-71.5X speed increase and outperforms XGBoost with speeds up to 11.8X. The effectiveness of the presented optimization has also been evaluated, with SparkTree outperforming MLLib with 7.70% AUC gain and outperforms XGBoost with 5.77% AUC gain.

Keywords: Tree Ensembles, Learning to Rank, Distributed System

1. Introduction

Decision tree is one of the most widely used machine learning models in industry, because it can model non-linear correlation, get interpretable results and does not need extra feature preprocessing such as normalization. When combined with different loss functions, it can be used across a wide variety of domains for classification Rokach and Maimon (2008), regression Breiman et al. (1984) and ranking Zheng et al. (2008); Burges (2010). There exist many variants in the decision tree family when combined with ensemble techniques such as bagging Breiman (1996) and boosting Schapire and Freund (2012). Examples include CLS Earl B. Hunt (1966), CART Breiman et al. (1984), ID3 Quinlan (1986), C4.5 Quinlan (1993), SLIQ Mehta et al. (1996), Random Forest Breiman (2001), and Gradient Boosting Machine (GBM) Friedman (2000); Tyree et al. (2011) to name just a few. Especially, variants of the gradient boosting decision tree (GBDT) achieve state-

of-the-art results in many web applications. For instance, LambdaMART Burges (2010) (a boosted tree version of LambdaRank Burges et al. (2007)) won the Yahoo! Learning To Rank Challenge (Track 1) Chapelle and Chang (2011) and it is considered as one of the most important models used for search ranking in Microsoft Bing, Yahoo! and Yandex Schamoni (2012); and Facebook uses it He et al. (2014) to implement non-linear feature transformation for online advertisements. It is a waste if they are implemented in separated systems. Thus there is an increasing demand for an `integrated` learning framework that supports all loss functions and ensemble techniques. A single tree is fitted by recursively splitting the nodes starting from the root. Each node split needs to traverse all training data and find the best split among all split candidates. With an increasing amount of available `big data` that has hundreds of millions of samples and each sample having thousands of features, and the increasing amount of `big model` in which thousands of trees are fitted with each tree has hundreds of nodes, there is also an increasing demand for a `scalable` distributed training algorithm that exploits all dimensions of parallelism.

Consider two real web-scale applications (Section 3): search ranking and ads click prediction. Existing offers such as XGBoost Chen and Guestrin (2016) and MLlib Spark (2012) in Apache Spark Zaharia et al. (2012) do not meet our needs. First, search ranking is not supported since they do not support LambdaMART. Second, they are not scalable and even fail to fit the training data with 160M samples $\times$ 1,280 features. More specifically, without the support of the feature parallelism where the data is vertically partitioned in a feature-wise way (Section 4), they are not scalable to support a large number of features. The feature parallelism is particularly preferred in our search ranking case, where the number of samples is increased slowly since they are manually labelled by vendors, while features that comes with different syntactic combinations and semantic embeddings are easily increased Blei et al. (2003); Huang et al. (2013); Mikolov et al. (2013).

Simply adding these supports does not meet the `efficiency` requirement that needs to balance the training speed and accuracy. Instead, in this paper we present SparkTree which provides an integrated learning framework of tree ensembles and combines a series of system optimizations and algorithm innovations. More specifically, SparkTree makes the following contributions:

- It integrates the task parallelism, sample parallelism, and feature parallelism. It further presents 1-bit encoding for the feature parallelism that determines the left or right child where a sample falls into, which will reduce 8˜16X network I/O.

- It presents optimizations to improve both system performance and model accuracy, including: 1). Pre-binning being applied once before training, 2). The largest bin is excluded for histogram construction to exploit the non-uniform histogram distribution and the corresponding statistics are recovered later during gain calculation. 3). tree is first top-down split layer by layer for more task parallelism, followed by a bottom-up greedy merge for better model accuracy.

- SparkTree is implemented on Apache Spark Zaharia et al. (2012), and the evaluation against production workloads indicates that SparkTree outperforms MLlib with 8.3X-71.5X speed increase and XGBoost with up to 11.8X speedup. The effectiveness of presented optimization is evaluated.

The rest of the paper is organized as follows. Section 2 describes the GBDT workloads in production. Section 3 provides a brief primer on the serial learning of tree ensembles and the optimizations we present. Section 4 describes the distributed workflow in SparkTree, followed by

describing the new tradeoff we make. Section 5 presents the task parallelism that first splits the tree layer by layer and greedily prunes the worst splits by bottom-up merging. The evaluation is described in Section 6, followed by a discussion in Section 7. Related work is listed in Section 8 and we conclude in Section 9.

2. Application

In this paper, we focus on two web applications of GBDT in our production, learning to rank in web search and ad click prediction for online advertisements.

2.1 Learning to Rank

Consider the ranking problem in web search in which given an input user query, learning to rank Chapelle and Chang (2011) essentially learns a function that can predict the relevance value for document candidates, and ranking is thus converted to simply sort the candidates according to the predicted scores. Each $(query, document_i)$ pair in the training example is represented as a feature vector (x_i) and the relevance grade y_i that is manually determined by relevance expert. There are several (usually five: excellent, good, average, fair, and poor) grades for y that represents different levels of relevance between the query and the document. There are three categories of features: 1). features modeling the query that may include the number of terms, the TF-IDF value of each term, different kinds of term embeddings such as topic Blei et al. (2003), word2vec Mikolov et al. (2013) and DSSM Huang et al. (2013), the expanded queries, and query segments, etc.; 2). features modeling the web document that may include PageRank value, the number of distinct anchor texts, the language/region identity of the document, and the document class, etc.; and 3). features modeling the query-document correlation that describes the matching between the query and document. They may include the frequency of each query term in the document title and anchor-texts, and the embedding similarity between the query and document title, etc.; We choose to use LambdaMART Burges (2010) that is considered as state-of-the-art learning to rank algorithms Chapelle and Chang (2011). It integrates the λ computing in RankNet and LambdaRank with a boosting decision tree (MART) model that generates additive ensembles of regression trees. λ can be considered the approximated gradient of NDCG Wikipedia that is actually non-differentiable. LambdaMART outperforms GBRT Tyree et al. (2011) which just approximates the root mean squared error rather than NDCG. This loss function in GBRT makes a point-wise training, i.e., query-document pairs are exploited independently. However as a comparison, LambdaMART directly optimizes the list-wise Chapelle and Chang (2011) information retrieval measures such as NDCG.

2.2 Ads Click Prediction.

In search advertising, paid advertisements may be placed by a search engine on a search result page. Ads "click probability" measures the probability that a user will click on the ad. The prediction is usually treated as a classification problem in machine learning. The accuracy of click prediction largely affects the revenue, thus playing an important role in monetization. Beyond commonly used machine learning models such as logistic regression McMahan et al. (2013), additive boosting trees such as GBDT also demonstrates benefits in deriving high-order feature conjunctions to reduce error residuals. Specifically, ad CTR in Facebook He et al. (2014) uses GBDT for non-linear feature transformation and feeds them to LR for the final prediction; and Microsoft Bing Ads team Lin et al.

(2017) uses GBDT to boost the prediction of a deep neural network (DNN) model, i.e., initializing the sample target for GBDT with the prediction score of DNN. The training data is collected from an ad impressions log, with each sample (x_i, y_i) representing whether or not the impressed ad allocated by the ad system has been clicked by a user. The output variable y_i is 1 if the ad has been clicked, and $y_i = 0$ otherwise. The input features x_i consist of different sources that describe different domains of an impression. 1). `query` domains that include query term, query classification, query length, etc. 2). `ad` domains that include ad ID, advertiser ID, campaign ID, and the corresponding terms in ad keyword, title, body, URL domain, etc. 3). `user` domains that include user ID, demographics (i.e., age, marriage, interest, profession, etc.), and user click propensity Cheng and Cantú-Paz (2010), etc. 4). `context` domains that describe date, and location. and 5). `cross` domains among them, e.g., $QueryId\_X\_AdId$ (X means cross) that crosses the user ID with the ad ID in an example.

3. Sequential Training of Decision Tree

In this section, we first briefly describe the sequential splitting process of a single decision tree, followed by presenting the optimizations $SparkTree$ uses.

3.1 Decision Tree

Given the training dataset $D = \{(\mathbf{x_i}, y_i); i = 1, \ldots, n\}$, a decision tree algorithm learns a regression, classification or ranking model $f \in \mathcal{F} : \mathcal{X} \rightarrow \mathcal{Y}$ by minimizing the loss objective on the training data. Model $f(\mathbf{x}) = w_{q(\mathbf{x})}$ consists of an independent tree structure q that maps a sample $\mathbf{x}$ to its corresponding leaf index ($q(\mathbf{x})$), and w_i represents the prediction value of i-th leaf.

The training process of a decision tree essentially constructs a binary tree by recursively splitting the node candidates (current "leaf" nodes) starting from the root node, until the number of leaf nodes or the depth of the tree reaches the regularized limit. Taking all samples on the current leaf, the splitting process includes two major steps: ***FindBestSplit*** and ***ApplySplit***. First, ***FindBestSplit*** enumerates all the possible split thresholds of all features, and selects the best split $(feature, threshold)$ with the largest gain. Secondly, ***ApplySplit*** moves a sample into the left child otherwise right child if its value of the selected feature is less than the threshold. Given a split of $parent$ node with two children, $left$ and $right$, the *gain* is generally defined as $gain = score_{left} + score_{right} - score_{parent}$, where the *score* of a node is calculated based on samples residing on it. The specific score function varies with different loss functions and optimization algorithms, e.g. information entropy and variance of the samples for classification and regression, respectively.

In the $FindBestSplit$ step, it has to enumerate all possible splits and select one split which has the best gain. In order to compute the gain of all possible splits in an *exact greedy algorithm* Chen and Guestrin (2016), for each feature, it has to sort the samples according to attribute values and calculate the statistics for each possible split. However, it is too expensive and even prohibitive to do this sorting in a distributed environment. Considering the expensive cost of an exact greedy algorithm, XGBoost usually resorts to an *approximate algorithm* Chen and Guestrin (2016), which is based on data histogram Ben-Haim and Tom-Tov (2010); Tyree et al. (2011); Gu et al. (2015). An approximate algorithm first buckets continuous feature(attribute) values into discrete ***bins***, then aggregates the statistics of each bin to construct a ***histogram***. Each feature has a histogram with b bins, and the i^{th} bin (b_i) consists of the *addictive* statistics of samples in that bin. The histogram is then computed by traversing all samples on the node to be split, and for each sample, the feature

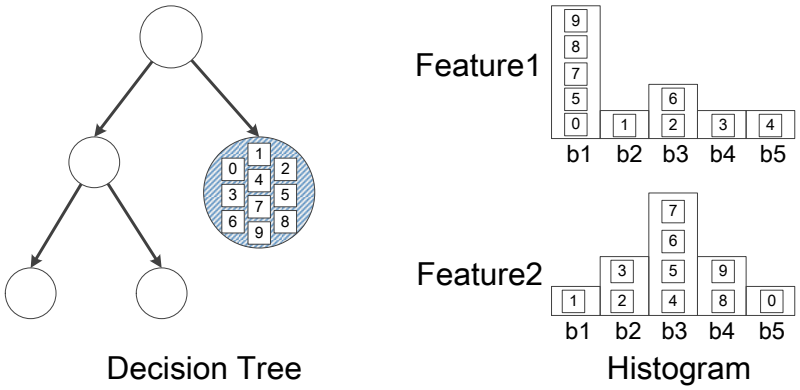


Figure 1: Histogram construction.

value is used to locate the corresponding bin of the histogram, and the statistics in that histogram's bin are accumulated accordingly, specifically for LambdaMART Burges (2010); The statistics includes a triple of $(count, \lambda, \lambda^2)$, where $count$ is the number of samples and λ is the gradient of a loss function. For example, Figure 1 illustrates the simple histogram construction process. Lastly, a *gain* is computed for each split candidate based on the constructed histogram, and the best split (a triple of $(feature, threshold, gain)$) is selected such that *gain* is maximized if data is split based on that $feature$ with the specific $threshold$.

After completing a node split, the two children are added to a task queue that holds the node candidates that is eligible for splitting, i.e., the number of samples in that node is larger than a threshold. This process continues until the tree is completed when the number of leaves reaches the given regularized upper bound or no leaves can be split. The *leaf value* of a leaf node is usually computed based on the average target value of samples in that node. With regard to the gradient boosting tree, once a tree is completed, we need to compute the new gradient (residual) for all samples that will be used in the next tree, thus multiple trees should be trained one by one. As a comparison, different trees can be trained in parallel in a Random Forest such that each tree is fitted against part of the training samples and features that are stochastically selected.

3.2 Global Approximation

The binning cost for a feature value is $O(\#bins)$ or even more if the value needs to be sorted first. It is thus time consuming if data needs to be re-binned for each newly split node, which is *local* and *dynamic*. However, it is unnecessary to pay the binning cost for each tree, since they have the same binning result [1]. Therefore, we instead choose to pre-bin the data once for a tree before its training, which is *global* and *static*. SparkTree takes one step forward further that we actually only do *pre-binning* once before the whole training. This not only saves the binning cost but also saves the storage for training data since we replace the original feature value (4 bytes of a float value) with the corresponding bin id (1 byte or 2 bytes of a char or short value). The possible accuracy loss due to pre-binning can be remedied by the boosting process and can be compensated by more bins or better binning algorithms Chickering et al. (2001).

1. Actually they also have the same histogram for all root nodes.

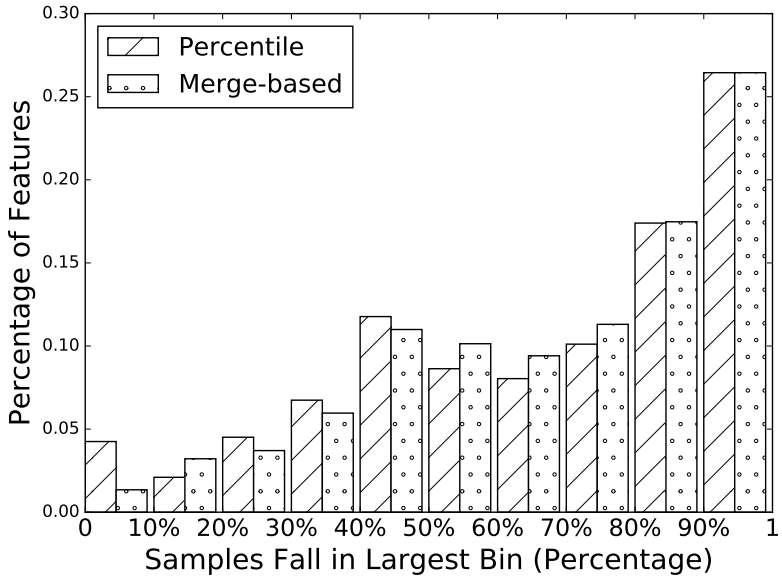


Figure 2: Non-uniform distribution.

3.3 Non-uniform Distribution Awareness

We have observed that the number of samples falling into each bin has non-uniform distribution in most histograms, i.e., the largest bin contains a large portion of samples. Although there are various binning approaches, e.g., percentile binning allocates each bin to have the equal-width of value range, and greedy-merge binning first allocates each sample to a unique bin and then merges the nearest two bins, the non-uniform distribution is still a common phenomenon.

Figure 2 depicts the distribution of features by binning into 64 buckets (statistics based on LtR dataset in Section 6.1). The X-axis is the percentage of samples that fall into the largest bin, and Y-axis is the percentage of features. For example, there are about 26.4% of features whose largest bin contains more than 90% of samples. In average, the largest bin contains about 67.12% of samples in the percentile-based binning method, and about 69.0% in greedy merge method. Generally, this is a common phenomenon of data in decision trees. Such non-uniform distribution comes from inherent data sparsity: missing values, frequent zero entries in the statistics, one-hot encoding, and a specific binning algorithm. Note that about 80% of the largest bin is the first bin, since there are many one-hot encoding features and a large amount of the feature values are zero. Therefore, we can leverage this non-uniform distribution of data in bins to optimize computation in histogram construction.

Samples' non-uniform distribution indicates skew computation in histogram construction. Figure 1 depicts how histograms are constructed. Each feature corresponds to a histogram. For each feature, all the samples on the current node are processed so that the feature value (a bin id after pre-binning) is used to locate the corresponding bin of the histogram, and the statistics in that histogram's bin are updated. Therefore, it takes $O(n)$ time complexity to construct the histogram of one feature where n is the number of samples in one single node; and for each bin of the histogram, the time complexity is proportional to the number of samples it accommodates. Due to samples' non-uniform distribution in the histogram shown in Figure 2, each bin of the histogram has a dif-

ferent computation cost. Most of the bins have little computation costs, and only the largest bin is heavily accommodated and costs significant computation. Optimization opportunity thus exists by eliminating the constructing cost of the largest bin in the histogram, which is indirectly computed by subtracting the combined statistics of all other bins from the corresponding total value in the current node. The total statistics values are summed during the node splitting process ($ApplySplit$) when $sampleToNode$ information is computed, and the total values include the number of samples residing at the current node, and the total summed statistics of these samples. Therefore, SparkTree does not store the sample values that are located in the largest bin, which indirectly sparsifies the training data ($featureValue$). This is especially useful when the training data is originally dense. Moreover, the cost of histogram construction is also largely saved so that about 70% of the workload is reduced. For example, the histogram h_{ij} (bin b_j in node n_i) is updated with $h_{ij} = total_i - \sum_{k \neq j} h_{ik}$ where $total_i$ is the total histogram sum of node n_i. Note that the total sum ($total_i$) of node i can be pre-computed when computing the $sampleToNode$ information in the $ApplySplit$ phase of the last split [2].

This optimization for histogram construction can save both computation and storage without extra cost. Additionally, the sparsity-awareness optimization that always skips the first bin can be considered a special case. Consider that different features have different degrees of non-uniform distribution, such optimization would result in a possible load imbalance, especially when data is feature-wise partitioned. We offset this side effect by modifying the partition algorithm that first sorts the features based on the degree of non-uniformness and partitions features to workers in a round robin manner.

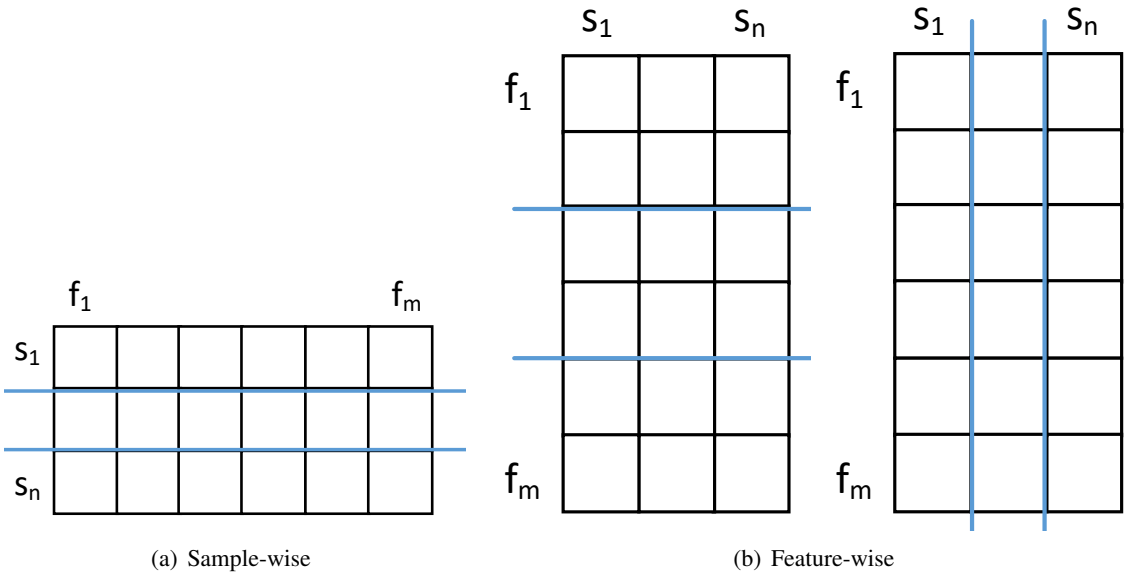
4. Parallelism Design

In this section, we first describe the overview of our distributed work flow, followed by reasoning our new designs with consideration for the balance of training efficiency and model accuracy. Here we focus on the splitting process of a single node n that has s samples and each sample has f features, each feature has b bins [3]. The splitting process includes two steps: 1). *FindBestSplit* to find the best split, 2). and *ApplySplit* to save the newly split result and compute $sampleToNode$ information that records the tree node that a sample resides on after a split. The specific distributed work flow depends on the choice of data partition.

4.1 Sample Parallelism

Sample-wise partition. In a sample-wise partition (*sample parallelism*), training data is horizontally partitioned across m machines (Figure 3(a)), i.e., all feature values of a sample are completely in a single partition, but all values of a specific feature are separated across partitions. Figure 4 depicts the distributed work flow of the sample parallelism. Given a tree node to be split, each worker first computes the local *incomplete histograms* based on its local samples that fall into that node, followed by shuffling the local histograms and aggregating the complete histograms with respect to a single feature and finding the *local* best splitting feature that has the largest gain; the best splits of all features are then aggregated to master (filled in gray color) and mater will find the *global* best splitting feature (*FindBestSplit*) with the largest gain. Since each feature has a histogram with b

2. It has actually been computed in $FindBestSplit$ phase of last split.
3. SparkTree chooses to use approximated histogram based algorithm to find the best split.



(a) Sample-wise (b) Feature-wise

Figure 3: Data partition. Blue line is the partition boundary.

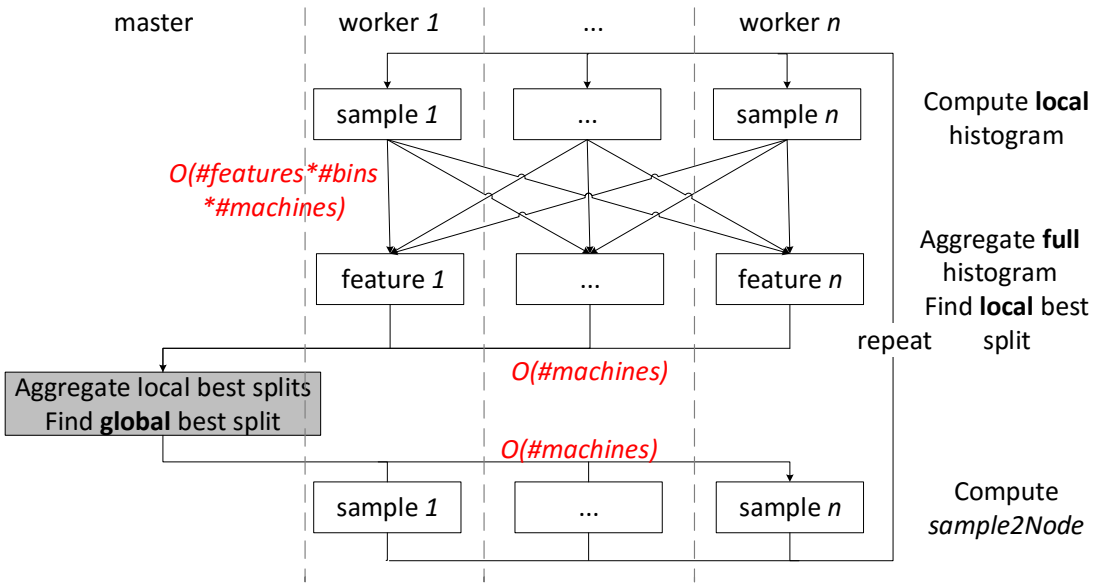


Figure 4: Workflow in the sample parallelism (horizontal).

bins, and the i^{th} bin (b_i) is a triple $(counts_i, \lambda_i, \lambda_i^2)$ with 12 bytes. [4] The shuffling between workers introduces $O(12 * f * b * m)$ (the size of histograms) network communication cost. Afterward in the *ApplySplit* step, the master records the newly split tree structure (model) and broadcasts the best splitting information (splitting feature and splitting threshold) to all workers, with each worker independently computing the $sampleToNode$ information for samples in its local partition. Note that the cost of aggregating local splits and broadcasting global split is $O(m)$ which is negligible.

Two-phase approximation. To reduce the communication cost in a sample-wise partition, Meng et al. (2016) proposes an approximate algorithm PV-Tree to vote for the best split with incomplete histograms. This approximation has two-phase communication between worker and master. In the first phase, the worker only transfers its local top k (a threshold) histograms to the master instead of all local histograms; The master then selects the top l splits according to the aggregated histograms; and in the second phase, the master sends the selected top l splits back to workers such that workers can provide feedback on the histograms of these l features to the master for the final decision. The network I/O is largely reduced since $k + l$ is much less than the number of features. However, this two-phase approximation actually trades the model accuracy for the reduced communication cost, and in production we are more conservative thus preferring an accurate split. Note that SparkTree also supports approximation techniques such as stochastic sampling of samples and features, which is used in random forest.

4.2 Feature Parallelism

Feature-wise partition. Training data is vertically partitioned (*feature parallelism*) so that each machine works on non-overlapping sets of features in parallel (Figure 3(b)), i.e., values of each feature for all samples are completely located in a single partition, but a sample is separated across partitions. The work flow of the feature parallelism is shown in Figure 5. In the *FindBestSplit* step, each worker first computes *complete histograms* for the features in its partition, then computes the best split threshold for each feature accordingly, and lastly selects the *local* best feature split with the largest gain. Only the local winner is then aggregated by the master to make the decision of the global best split. This aggregation incurs negligible communication cost with $O(m)$ across m machines.

Transposed feature-wise copy. The $sampleToNode$ information in the feature-wise partition can only be computed by a single worker who owns the selected best feature, which largely limits the parallelism since other workers need to stall and wait. To avoid the sequential execution, SparkTree provides another *transposed* copy of the feature-wise partitioned data [5] (Figure 3(b)). Thus $sampleToNode$ can be computed by all workers in parallel. This extra space overhead is acceptable in most cases (45GB and 52GB in our two production applications). The local $sampleToNode$ information is then aggregated (concatenated) by the master which then broadcasts back to all workers, which incurs communication cost with $O(s)$ (the number of samples). It is noteworthy that the hybrid partition (blocking) approach that partitions training data both horizontally and vertically at the same time has never been used, since it brings no benefits but inherits all of the disadvantages.

One-bit encoding. The node identifier in $sampleToNode$ is usually encoded with 8 bits when the number of leaf nodes is less than 256, otherwise it is encoded with 16 bits (less than 2^{16} nodes).

4. It could be a quad with additional elements if second order gradient is used to compute the gain Chen and Guestrin (2016); Burges (2010).
5. Training data is pre-transposed before training with a little cost, since we do not transpose element in a partition and data is still in feature-wise layout in each partition.

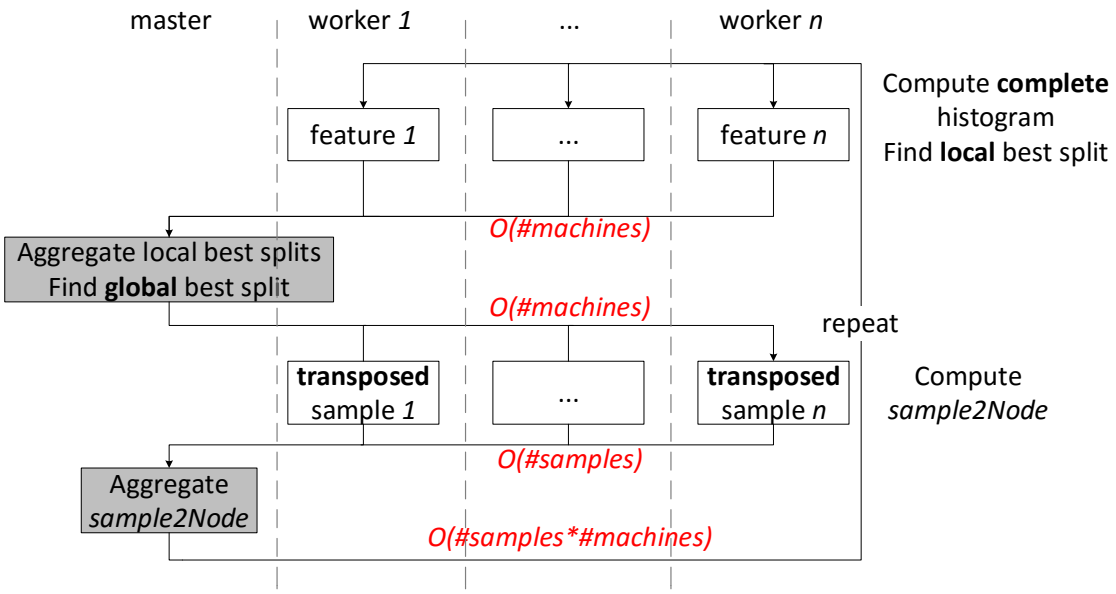


Figure 5: Feature-wise partition (vertical).

However, 1 bit is enough to encode the *change* after the split. We use 1 bit to record if sample falls into the left child or otherwise. The new $sampleToNode$ information can be recovered from the old state with 1-bit encoded update. In this way, the network I/O can be reduced by 8 or 16 times.

4.3 Task Parallelism

Beyond the the data parallelism, SparkTree also supports another dimension of parallelism, the *task parallelism*, in which multiple tree nodes are split independently in parallel based on the part of the training data that falls into the node, thus the overall system performance can be largely improved. The communication cost is high if only the task parallelism is used Tyree et al. (2011) since the training data needs to be moved from machine to machine multiple times during the training process. Instead, we choose to integrate the data parallelism (sample or feature parallelism) with the task parallelism to avoid costly training data transferring, by just maintaining the $sampleToNode$ information and a task queue that keeps the tree node candidates ("current" leaf nodes) that are eligible to be split. During the *FindBestSplit* step, the worker computes histograms for all available nodes (n) at the same time, and the total histogram size is increased by n times to be $f * b * n$. In addition, the communication cost in sample-wise partition is also increased n times. To bound the resource usage, SparkTree supports to control the upper bound of the task parallelism.

When the task parallelism is integrated, the histogram construction can also leverage the non-uniform distribution for computation savings. Figure 6 depicts the process for one feature for simplicity. Each feature is associated with two input vectors: $featureValue$ that keeps the correspond-

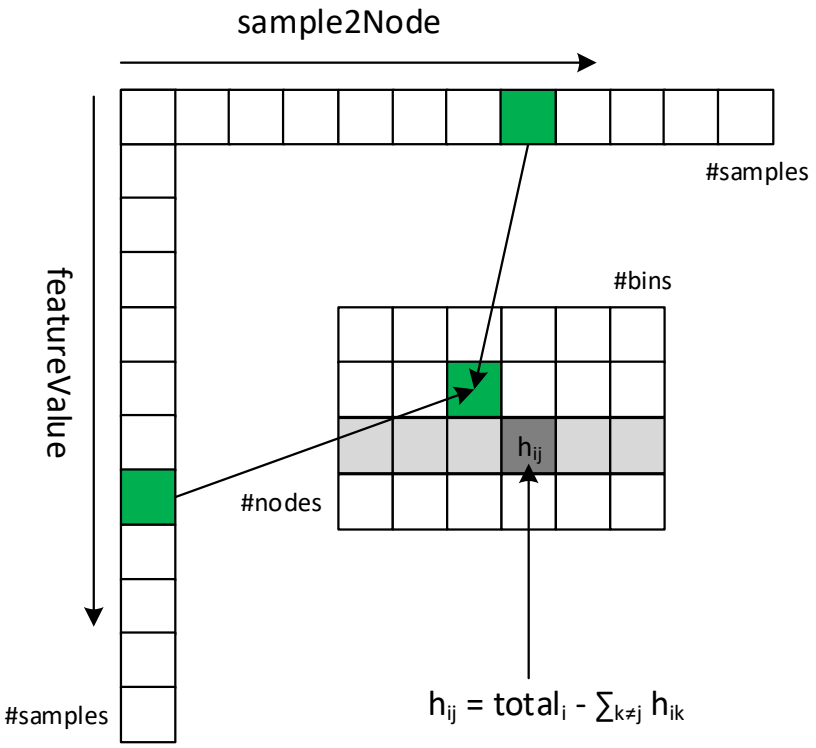


Figure 6: Histogram construction in the task parallelism.

ing bin id for each sample [6], and $sample2Node$ that records the corresponding node id for each sample. The histogram is represented as an array with the size of $bins$ (a matrix here since multiple nodes are processed in parallel.). For the i^{th} sample, $sampleToNode[i]$ is used to locate the histogram of the corresponding node and $featureValue[i]$ is used to locate the specific bin in that histogram (green in Figure 6). The histogram h_{ij} (the largest bin b_j in node n_i) in gray in Figure 6 is updated with $h_{ij} = total_i - \sum_{k \neq j} h_{ik}$ where $total_i$ is the total histogram of node n_i.

4.4 Design Tradeoff

In this section, we describe the design tradeoff on the right choice of data partition that has a significant impact on system performance. Each choice has its own suitable scope and the right choice depends on the specific input and the specific tradeoff system makes. Table 1 lists comparisons between the sample-wise partition and feature-wise partition. As described above, the network I/O of a feature-wise partition is s bytes that is agnostic to the number of features, bins and nodes that are split in parallel. As a comparison, sample-wise partition has a communication cost of $f*b*3*4$ bytes (each bin is a tuple with at least 3 elements [7] and each element has 4 bytes) that is agnostic to the number of samples. The tipping point is $s > f*b*12$ if the task parallelism is disabled, otherwise, the tipping threshold is $s > f*b*n*12$ where the balance inclines n times towards the

6. Value of different samples for the same feature is located together.
7. It could be tuple with four elements if second order gradient is used to compute the gain Chen and Guestrin (2016); Burges (2010).

Attributes	Sample-wise	Feature-wise
$\#features$ dependent	yes	no
$\#bins$ dependent	yes	no
$\#nodes$ dependent	yes	no
$\#samples$ dependent	no	yes
optimization	two-phase approximation	1-bit encoding
approximation	yes	no
network I/O	$12 * \#features * \#bins * \#nodes$	$\frac{1}{8}\#samples$

Table 1: Comparison between sample-wise and feature-wise partition. A single bin has 12 bytes, and the optimization with 1-bit encoding can reduce the size with 8 times.

feature-wise partition. With the presented one-bit encoding, the communication cost in the feature parallelism is further reduced to $\frac{s}{8}$ from s bytes.

As a consequence, unlike conventional wisdom, the strength scope of the feature-wise partition is enlarged significantly by $12 * 8 * n$ times and we find that in many production workloads the right choice should be the feature-wise partition. Consider a case in ads click prediction that has a large number of samples (300M) and a small number of features (1500), 200 bins and each tree has 400 leaf nodes. The first step would be to choose the sample-wise partition since $300M > 12 * 1,500 * 200$. However, when we enable the 1-bit encoding and the task parallelism (with the averaged 32 nodes split in parallel), the right choice should be the feature-wise partition. Besides, there are two other design considerations that further tip the balance toward the feature parallelism: 1). The number of samples could actually be further reduced if we do down-sampling He et al. (2014) to exclude a large portion of negative samples, and the number of bins would be enlarged for better accuracy. 2). It is also noteworthy that the feature-wise layout in a partition is more CPU cache friendly and more space efficient so that we can construct the histogram of all features one by one rather than interleaving them together with a large amount of random access.

5. Task Parallelism Optimization

In this section, we present novel optimizations to improve the model accuracy when the task parallelism is enabled.

Bottom-up based merging VS. top-down based splitting. With respect to the task parallelism shown in Figure 7, there are two extreme cases: 1). node is sequentially split one by one (left part); 2). node is split in a layer-by-layer fashion (right part), i.e., all leaf nodes that are eligible to be split are split at the same time. One-by-one is considered as a greedy algorithm that always picks the node with the largest split gain (the gain corresponding to its best split). Thus the one-by-one strategy is considered to have better accuracy than layer-by-layer, though the latter has much better system performance. However, this is not always true since the one-by-one approach does not guarantee to be optimal since it only looks forward one additional depth to make the decision. Take the case shown in the middle part of Figure 7, the split of node 2 should have been better than the split of node 1, while the one-by-one method chooses to split node 1 since it cannot compare the split gain of the children or even the descendants of current split candidates.

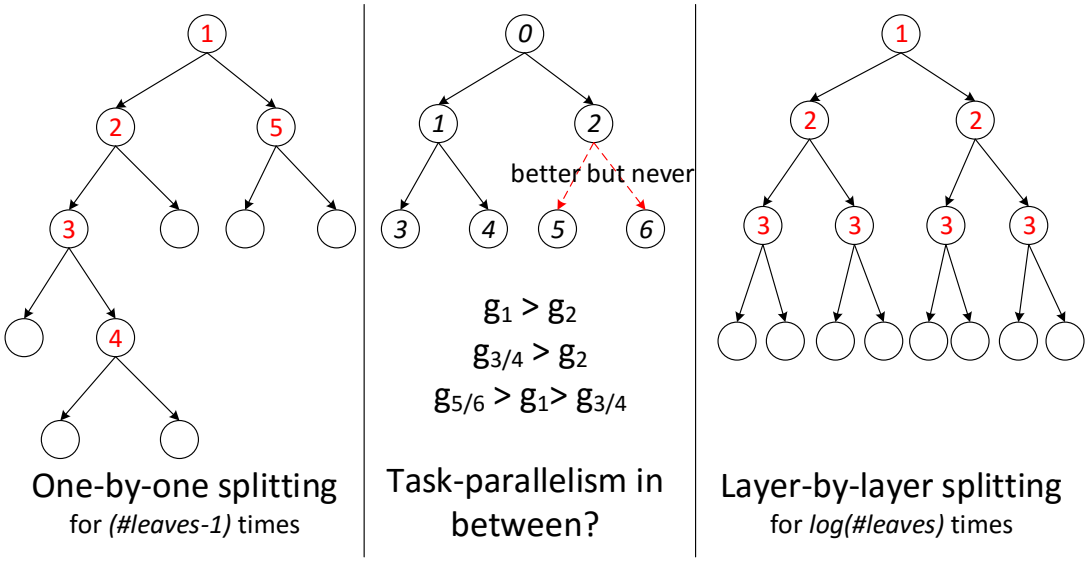


Figure 7: One-by-one splitting (left) versus layer-by-layer splitting (right). The number in red is the splitting order. The middle part illustrates how greedy sequential splitting is suboptimal.

A better task parallelism strategy exists between the one-by-one and layer-by-layer method that has more parallelism than one-by-one and better accuracy than layer-by-layer or even one-by-one. SparkTree achieves this by first doing a top-down layer-by-layer split but with one or two more layers, followed by `greedily merging` the bottom subtree one by one in a `bottom-up` way until the leaf nodes reach the regularized number. A `bottom subtree` (BST) only contains 3 nodes ($left$, $right$, and $parent$), and is selected for merging among all candidates if the split gain ($score_{left} + score_{right} - score_{parent}$) is the smallest (Section 3). The merge simply removes two children and only keeps the parent. The specific algorithm is shown in Algorithm 1. It is easy to prove that this algorithm is optimal , i.e., the summed scores of the resulting leaves minus the score of the root note that is the largest. We can also prove that it is even better than the one-by-one method if the tree is first split layer-by-layer with the same depth as the tree split using the one-by-one method. In practice, we usually split only $\sqrt{leaves} + 2$ layers. This is a tradeoff for which we only pay $\frac{\sqrt{leaves}+2}{\sqrt{leaves}}$ times of the layer-by-layer method but with much better accuracy, and in most of cases it also has better accuracy than the one-by-one method. Note that the merge is done in the master, with $O(2^d - l)$ time complexity where d is the depth of the input tree and l is the regularized leaf number.

6. Evaluation

We have built SparkTree on top of Apache Spark Zaharia et al. (2012) which improves the distributed data-parallel systems such as MapReduce and Hadoop by providing a Resilient Distributed Datasets (RDDs) abstraction. Spark largely simplifies our engineering effort with only 5,377

Algorithm 1 Bottom-up based greedy merging algorithm.

Input: A complete tree T with 2^d leaf nodes.
Output: A pruned tree with l leaf nodes.
procedure BOTTOMUPGREEDYMERGE
    for each $leaf$ in leaves of T **do**
        $parent = getParent(leaf)$
        $sibling = getSibling(leaf)$
        if $sibling$ is a leaf **then**
            $BSTSet.AddBST(leaf, sibling, parent)$
    $steps = 2^d - l$
    while $steps > 0$ **do**
        $min = 0, target = null$
        for each $bst \in BSTSet$ **do**
            $(left, right, parent) = getNodes(bst)$
            if $min > score_{left} + score_{right} - score_{parent}$ **then**
                $min = score_{left} + score_{right} - score_{parent}$
                $target = bst$
        $(left, right, parent) = getNodes(target)$
        remove $left$ and $right$
        mask $parent$ as leaf node
        $root = getRoot(parent)$
        $sibling = getSibling(parent)$
        if $sibling$ is a leaf **then**
            $BSTSet.AddBST(parent, sibling, root)$
        $steps- = 1$

lines of Scala code in our implementation, and the essential functionalities in distributed computing such as data partition, network communication, fault tolerance, task scheduling and monitoring have already been provided by Spark. Note that the proposed techniques in SparkTree are generic and can also be applied to XGBoost or other tree ensemble learning systems. We have shared our ideas with our colleagues and apply the techniques in LightGBM lig (2016) which achieves high performance. Our implementation reuses many of the codes in MLlib Spark (2012), a machine learning library in Spark that already has the support of tree ensemble learning. What we add are LambdaMART support, feature-wise partition with 1-bit encoding (Section 4) and optimizations presented in Section 5. We skip the implementation details and readers can refer to the source code in `https://github.com/cloudml/sparktree`.

6.1 Evaluation Setup

In this paper, we focus on two web applications of the gradient boosting tree in our production, learning to rank and ads click prediction (Section 2). Their dataset sizes and training configurations are listed in Table 2. In the first tree, sample scores are always initialized to 0.0, and we set the shrinkage rate as 0.1.

Dateset	Samples	Features	Bins	Tree number	Leaf nodes
LtR	10M	4,095	1,024	1,000	256
AdsCTR	130M	320	128	1,000	256

Table 2: Dataset.

Cluster setup. The evaluation has been conducted on a Spark cluster which has 10 homogeneous computing nodes that are connected via 40Gbps Infiniband network and each node has 16 2.40GHz Intel(R) Xeon(R) CPU E5-2665 cores and 128GB memory. There is 1 master configured with 5GB memory and 10 workers configured with 80GB memory. Data is in 160 partitions by default.

6.2 Comparison against XGBoost and MLlib

We have compared SparkTree with two widely used systems, XGBoost and Spark MLlib. We use two versions of XGBoost for comparison, one is native XGBoost on Yarn and another is XGBoost4J (integrated with Spark). XGBoost4J provides the Java/Scala API calling the core functionality of XGBoost library, and has been integrated with Spark. The communication among workers in XGBoost4J goes through Allreduce synchronization in native XGBoost that has much better performance than the shuffling service in Spark. MLib, XGBoost, XGBoost4J and SparkTree share the same settings that their data has the same number of partitions, and each tree has a depth of 8 that is trained via a histogram based algorithm. Note that XGBoost and MLlib suffer high cost on data preprocessing (binning) and we exclude them in comparison.

Performance. The performance comparison is first conducted based on different configurations.

Varied Feature Number. Only the number of features varies from 320 to 4,800 (simulating a common feature number in practice), with a fixed number of samples (10M LtR dataset) and bins (1,024). The feature number is changed by random sampling. Figure 8 depicts the comparison results on the averaged training time per tree. SparkTree outperforms XGBoost with about 4.1X speed increase and outperforms MLlib with about 46.6X speed increase when the number of features is 1,600. As the feature number increases, SparkTree scales smoothly with sub-linear time increase, XGBoost scales well but with linear time increase, and MLlib is the worst since its time cost increases super-linearly. XGBoost increases super-linearly when the number of features is larger than 3,200 because it has to use external memory to assist computation, while SparkTree utilizes non-uniform distribution awareness to store feature values. The major reason that SparkTree achieves superior performance is that network I/O in SparkTree is feature-number independent since a feature-wise partition is used, but XGBoost and MLlib suffer from a feature-number dependent network I/O.

Varied Bin Number. The number of bins varies from 64 to 1,024, with fixed number of samples (10M LtR dataset) and features (1,280). Figure 9 depicts the performance comparison result. SparkTree outperforms XGBoost/XGBoost4J/MLlib with about 2.8X/4.4X/35.2X speed increase respectively when the number of bins is 1,024. And the time in SparkTree only increases a little bit (16s versus 22.7s) when the bins are increased to 1,024, while it increases by about 3.5X in XGBoost (17.4s versus 64.2s) and MLlib (275.4s versus 798.6s). Again, this is because the network I/O is independent of the bin number in the feature-wise partition, and the increased overhead in SparkTree is mainly an increase in memory cost and computation due to the larger histogram.

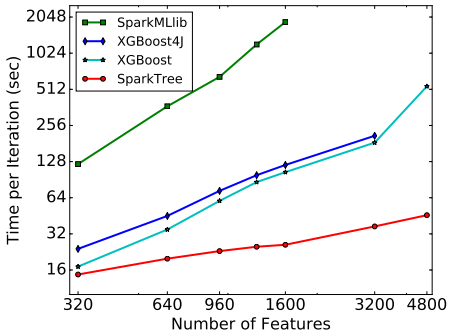


Figure 8: Varied feature number.

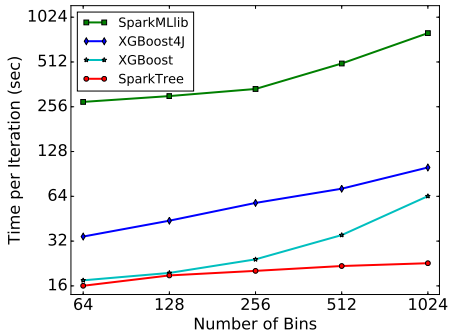


Figure 9: Varied bin number.

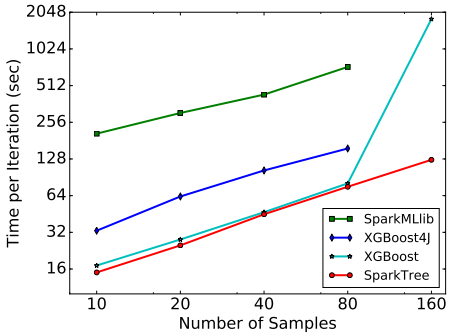


Figure 10: Varied sample number.

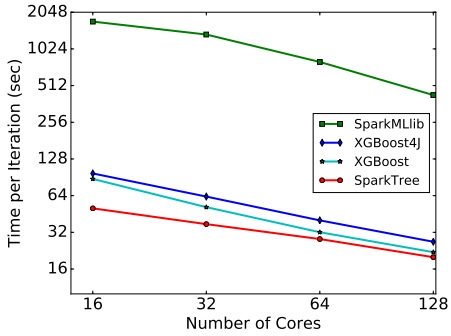


Figure 11: Varied core number.

Varied Sample Number. The number of samples varies from 10M to 320M (AdsCTR dataset), with a fixed feature number (1,280) and bin number (128). The result is depicted in Figure 10. SparkTree scales smoothly due to the optimization of non-uniform distribution, which saves both computation and memory. When the number of samples exceeds 160M, XGBoost utilizes out-of-core computation but XGBoost4J and MLlib fails. Although the feature values are dense, SparkTree still largely benefit from the memory optimization using non-uniform distribution (Section 3).

Varied Core Number. The number of cores varies from 16 to 128, with fixed sample number (10M LtR dataset), feature number (1,280) and bin number (128). The result is depicted in Figure 11. The evaluation indicates that SparkTree has the similar scalability as XGBoost. The speedup is sublinear because network I/O increases linearly, even though computing cost decreases linearly.

Accuracy. The accuracy comparison is against AdsCTR dataset listed in Table 2 since XGBoost and MLlib do not support LambdaMART. 70% data is for training and the rest is for testing. In all experiments, we boost 1,000 trees with a shrinkage of 0.1. MLLib and XGBoost have a maximum depth of 8, and SparkTree is regularized to have 256 leaf nodes from a maximum depth of 10. The AUC comparison for MLLib, XGBoost, and SparkTree is listed in Table 3. SparkTree achieves better accuracy, outperforming MLlib with 7.714% accuracy gains and outperforms XGBoost with

5.773% gains. The gain is mostly from bottom-up merging. There will be a detailed analysis will be described in the following subsection.

	MLLib	XGBoost	SparkTree
Training AUC	0.7779	0.8060	0.8186
Test AUC	0.7739	0.7881	0.8336
Test AUC gain	7.714%	5.773%	-

Table 3: AUC comparison of MLLib, XGBoost, and SparkTree.

6.3 Optimization Evaluation

We now evaluate the presented optimizations: exploiting non-uniform histogram distribution and bottom-up based greedy merging.

Non-uniform histogram distribution. The end-to-end system performance has been achieved about 34.3% (18.4s versus 28s) in greedy-merge based pre-binning and 31.2% (19s versus 27.6s) in percentile pre-binning. The speedup is much larger in histogram construction in `findBestSplit` step, while the network I/O and computation workloads in `applySplit` step are still unchanged.

Bottom-up based greedy merging. We have evaluated the effectiveness of bottom-up based greedy merge on ads click prediction and search ranking (Table 2). Layer-by-layer splitting is set with a maximum depth of 8, one-by-one splitting sets 256 leaf nodes, and bottom-up merging sets a maximum of 256 leaf nodes from 10 depths. The results are shown in Figure 12 and Figure 13, respectively. Compared with the layer-by-layer splitting, bottom-up achieves 7.66% AUC gain and 1.35% NDCG gain, which derives from 2 more depths of search space to find the best split. Compared with the one-by-one splitting, bottom-up merging achieves neutral AUC gain and about 1.76% NDCG gain. Bottom-up merging has better accuracy because it makes the decision to split a node from the whole tree point of view, but the gain is limited since its search depth is just increased to a depth of 2 that is much less than the depth (even 100+ depths) in the one-by-one splitting.

7. Discussion and Future Work

In this section, we discuss several interesting points of both the system and the algorithm.

System optimization. We have identified several optimization opportunities that are difficult to implement in Spark or with possible accuracy loss, and they are saved for future work: 1). The $count$ in histogram in root node is the same across all trees, and making it unnecessary to recompute them again. However, it is inconvenient to keep them across stages for which RDD is the only choice but with poor performance. 2). The histogram in a larger child equals the corresponding histogram in root subtracts the histogram in the smaller child, i.e., we only require computing the histogram of the smaller child. Again, the inconvenience of keeping state across stages limits the implementation in Spark. 3). Spark lacks asynchronization support to overlap communication with computation. The computation in each stage is usually fast (1-2s), thus even a small network I/O would occupy a non-ignorable portion of time. CPU resources are under-utilized during synchronization when CPU is idle to wait its completion. 4). We have not implemented the two-phase approximation that could reduce the network I/O in a sample-wise partition (Section 4), since it would result in possible accuracy loss which is unacceptable in production. However, it is still worth attempting in

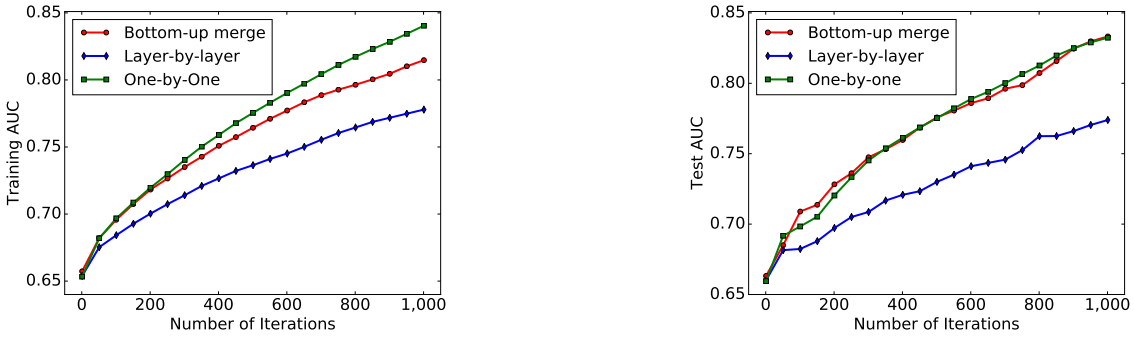


Figure 12: Effectiveness of bottom-up based greedy merge on Ads click prediction.

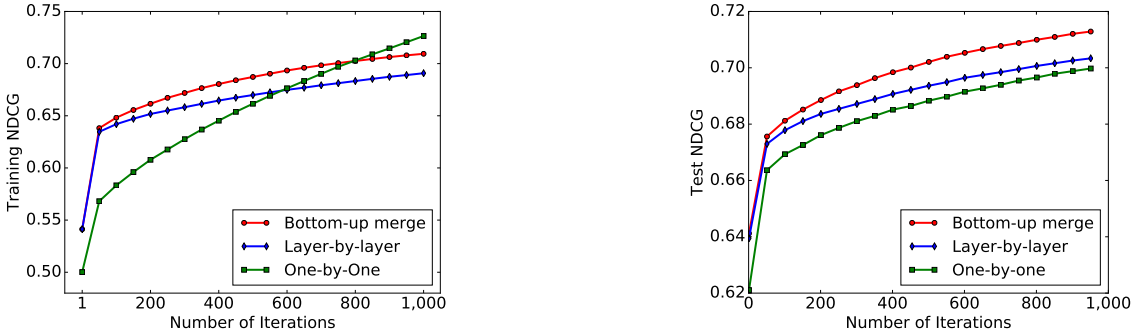


Figure 13: Effectiveness of bottom-up based greedy merge on search ranking.

future works for other applications. 5). We also consider abstracting tree fitting process as graph computing thus it can be implemented in GraphX. More specifically, the training data is modeled as a bipartite graph with two kinds of vertices, a sample vertex and a feature vertex. An edge exists from sample vertex to feature vertex if that sample has a non-zero value. Sparse data can be efficiently represented in a graph. The trained model and intermediate state are also annotated together with the graph: the feature vertex is annotated with the histogram attribute (an array with the length of bins), and the sample vertex is annotated with an attribute pair (target, node id), and the feature value is annotated as the edge attribute. During distributed execution, the graph is partitioned and each partition is executed in parallel. Beyond simple rectangle partition, the advantage of a graph is that it supports more flexible partitioning approaches.

Algorithm tuning. We have also tried several methods to improve model accuracy, but they have little gain or even worse accuracy. 1). We try the integration of bagging in boosting, i.e., each tree is fitted against part of the training samples and features that are stochastically selected. For LambdaMart, sampling is done with a query granularity that documents corresponding to a query that is either sampled all together or not sampled at all. 2). The tree complexity is adaptively controlled, i.e., different trees are regularized with different numbers of leaf nodes. 3). We have tried to give more weight to samples that are under-fitted, and less weight for samples that have already been fitted. A sample is considered to be fitted if the gradient (λ) is almost zero with little change in latest iterations. 4). A tree is regularized such that the same split is applied to all nodes in the same layer ndrey Gulin and Karpovich (2009). In our implementation, the split gains of a splitting candidate (the same feature and the same threshold) are aggregated together across all

nodes in the same layer, and we choose the split with the largest gain as the best. 5). Features that have been chosen over frequently or over rarely are penalized so that their gains are multiplied with a penalty rate with a value less than 1. Lastly, it would be interesting to apply adaptive learning rate techniques such as Momentum and AdaGrad Zeiler (2012), to tune the accuracy. A gradient boosting tree is also considered a gradient method such that the new fitting target for a sample is the negative gradient of loss function with respect to trees that have already been fitted. An adaptive learning rate has been proven to have better convergence in ordinal stochastic gradient descent (SGD). Specifically, we can add a change (learning) rate to control the change that takes place in the target direction of the negative gradient, and apply the same idea to adaptively tune the new sample target so that different samples have different target change rates in different iterations. Take LambdaMART Burges (2010) as an example, the computed λ for a document is treated as the negative gradient of NDCG loss. We reserve this direction future work.

8. Related Work

There are plenty of works on parallelizing and optimizing tree ensemble learning in the literature.

Parallel tree ensemble learning. Provost and Kolluri Provost and Kolluri contains a survey of existing approaches, along with the motivations for large scale tree learning. Srivastava et al. Srivastava et al. (1998) presents task parallelism, while Amado et al. Amado et al. (2001); Tyree et al. (2011) advocates data-parallelism. The data can be partitioned by features Freitas and Lavington (1997) or by samples Shafer et al. (1996). As a comparison, SparkTree seamlessly integrates all kinds of parallelism and supports all partition approaches. Moreover, the tradeoff between speed and accuracy is reconsidered in SparkTree such that feature-wise partition is more preferred, since the network I/O in a feature-wise partition is agnostic to the number of bins, features, and nodes split in parallel. Beyond simple adoption of feature-wise partition Ye et al. (2009), SparkTree enlarges its applicability by 1-bit encoding and gives the specific formula on the right choice. With regard to task parallelism, SparkTree chooses to adopt the breadth-first order Mehta et al. (1996); Shafer et al. (1996); Panda et al. (2009), rather than the depth-first manner Quinlan (1993).

Tree ensemble learning has been implemented on different systems, such as MPI Chen and Guestrin (2016), OpenMP Chen and Guestrin (2016), Hadoop Ye et al. (2009), MapReduce Panda et al. (2009) and Spark Spark (2012). Each choice has its own advantages and disadvantages. On the one hand, OpenMP and MPI are considered as the "assembly" language of distributed programming thus would have the best performance, while it lacks the fault-tolerance and has high programming complexity. On the other hand, data-parallel system largely simplifies the programming complexity for big data processing and its execution can be automatically scaled-out with fault-tolerance support. Ye et al. argue that Hadoop is not a good fit for tree learning Ye et al. (2009) since it suffers from high system overhead where persistent storage such as HDFS is used for data shuffling. This issue has been fixed in Spark and SparkTree chooses Spark since it has better performance than Hadoop and has much wider adoption in industry than MPI and OpenMP. Compared with the implementation in MLlib, we provide much better performance and scalability. Moreover, with optimization on the algorithm side, SparkTree even manages to outperform XGBoost that is implemented in native language and with an Allreduce based synchronization. Besides, compared with the completely integration framework in SparkTree, there is no support of feature parallelism and no support of LambdaMART in MLlib and XGBoost.

Optimized tree ensemble learning. SparkTree follows the same idea as works Chen and Guestrin (2016); Tyree et al. (2011) to approximate the exact split via histogram with different binning methods. Instead of local and dynamic binning, SparkTree only does global pre-binning once before the training starts. A similar idea can be found in XGBoost and MLlib. The optimization that exploits the non-uniform histogram distribution is similar to the sparse optimization of XGBoost. XGBoost treats missing values as empty values thus they are binned into the same bucket, making it unnecessary to count them in histogram and the best split is computed by traversing the histogram twice, from left to right or vice versa (Algorithm 3 in XGBoost Chen and Guestrin (2016)). As a comparison, SparkTree skips the maximum bin to indirectly sparsify the training data thus both the storage of training data and the complexity of histogram construction are reduced. Both of them will not affect accuracy, while SparkTree is supposed to have a larger performance gain since the bin contains the missing value is definitely less than the bins with maximum number of samples. There are also many works Patil et al. (2010); Kearns and Mansour (1998); Breiman et al. (1984); Quinlan (1993) on pruning the decision tree for better generalization. The largest difference in our context is that the pruning is constrained by the leaf number. However, CART Breiman et al. (1984); Quinlan (1993) uses expensive dynamic programming for pruning, and Bohanec and Bratko Bohanec and Bratko (1994) show that it is possible to compute in quadratic time. Our method has the same `local` characteristics as work Kearns and Mansour (1998) in which the decision to prune a subtree is based entirely on properties of that subtree and the sample reaching it.

9. Conclusion

In this paper, we present SparkTree which provides an efficient and scalable tree ensemble learning system. The proposed optimizations on computation and communication improve both training speed and model accuracy, which are generally useful in other system implementations such as XGBoost and LightGBM. With both algorithm innovation and system improvements, we demonstrate that this distributed data-parallel abstraction for boosting decision trees is not only feasible and beneficial, but also efficient and scalable.

References

LightGBM, light gradient boosting machine. `https://github.com/Microsoft/LightGBM`, 2016.

Nuno Amado, Joo Gama, and Fernando Silva. Parallel implementation of decision tree learning algorithms. In *Progress in Artificial Intelligence*. 2001.

Yael Ben-Haim and Elad Tom-Tov. A streaming parallel decision tree algorithm. *JMLR*, 11:849–872, March 2010.

David M. Blei, Andrew Y. Ng, and Michael I. Jordan. Latent Dirichlet Allocation. *JMLR*, 3:993–1022, 2003.

Marko Bohanec and Ivan Bratko. Trading accuracy for simplicity in decision trees. *Machine Learning*, pages 223–250, 1994.

L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*. Chapman and Hall, New York, 1984.

Leo Breiman. Bagging predictors. *Mach. Learn.*, pages 123–140, 1996.

Leo Breiman. Random forests. *Mach. Learn.*, 45:5–32, 2001.

Chris J.C. Burges. From ranknet to lambdarank to lambdamart: An overview. Technical report, June 2010.

Christopher J. Burges, Robert Ragno, and Quoc V. Le. Learning to rank with nonsmooth cost functions. In *NIPS*. 2007.

Olivier Chapelle and Yi Chang. Yahoo! learning to rank challenge overview. In *Proceedings of the Yahoo! Learning to Rank Challenge*, pages 1–24, 2011.

Tianqi Chen and Carlos Guestrin. XGBoost: A scalable tree boosting system. *CoRR*, abs/1603.02754, 2016. URL `http://arxiv.org/abs/1603.02754`.

Haibin Cheng and Erick Cantú-Paz. Personalized click prediction in sponsored search. WSDM, pages 351–360, 2010.

David Maxwell Chickering, Christopher Meek, and Robert Rounthwaite. Efficient determination of dynamic split points in a decision tree. In *ICDM*, 2001.

Philip J. Stone Earl B. Hunt, Janet Marin. *Experiments in induction*. Academic Press, New York, 1966.

Alex A. Freitas and S. H. Lavington. *Mining Very Large Databases with Parallel Processing*. Kluwer Academic Publishers, 1997.

Jerome H. Friedman. Greedy function approximation: A gradient boosting machine. *Annals of Statistics*, 29:1189–1232, 2000.

Rong Gu, Lei Jin, Yongwei Wu, Jingying Qu, Tao Wang, Xiaojun Wang, Chunfeng Yuan, and Yihua Huang. Characterization and parallelization of decision-tree induction. *Algorithms and Architectures for Parallel Processing*, pages 52–65, 2015.

Xinran He, Junfeng Pan, Ou Jin, Tianbing Xu, Bo Liu, Tao Xu, Yanxin Shi, Antoine Atallah, Ralf Herbrich, Stuart Bowers, and Joaquin Quiñonero Candela. Practical lessons from predicting clicks on ads at facebook. In *ADKDD*, pages 5:1–5:9, 2014.

Po-Sen Huang, Xiaodong He, Jianfeng Gao, Li Deng, Alex Acero, and Larry Heck. Learning deep structured semantic models for web search using clickthrough data. In *CIKM*, pages 2333–2338, 2013.

Michael J. Kearns and Yishay Mansour. A fast, bottom-up decision tree pruning algorithm with near-optimal generalization. In *ICML*, pages 269–277, 1998.

Xiaoliang Lin, Weiwei Deng, Chen Gu, Hucheng Zhou, Cui Li, and Feng Sun. Model ensemble for click prediction in bing search ads. In *WWW*, 2017.

H. Brendan McMahan, Gary Holt, D. Sculley, Michael Young, Dietmar Ebner, Julian Grady, Lan Nie, Todd Phillips, Eugene Davydov, Daniel Golovin, Sharat Chikkerur, Dan Liu, Martin Wattenberg, Arnar Mar Hrafnkelsson, Tom Boulos, and Jeremy Kubica. Ad click prediction: a view from the trenches. In *KDD*, 2013.

Manish Mehta, Rakesh Agrawal, and Jorma Rissanen. Sliq: A fast scalable classifier for data mining. In *EDBT*, pages 18–32, 1996.

Qi Meng, Guolin Ke, Taifeng Wang, Wei Chen, Qiwei Ye, Zhi-Ming Ma, and Tieyan Liu. A communication-efficient parallel algorithm for decision tree. In *NIPS*, pages 1271–1279. 2016.

Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *NIPS*, pages 3111–3119. 2013.

ndrey Gulin and Pavel Karpovich. Greedy function optimization in learning to rank, 2009. URL `http://romip.ru/russir2009/slides/yandex/lecture.pdf`.

Biswanath Panda, Joshua S. Herbach, Sugato Basu, and Roberto J. Bayardo. PLANET: Massively parallel learning of tree ensembles with mapreduce. In *VLDB*, 2009.

Dipti D. Patil, V. M. Wadhai, and J. A. Gokhale. Evaluation of decision tree pruning algorithms for complexity and classification accuracy. *International Journal of Computer Applications*, 2010.

Foster Provost and Venkateswarlu Kolluri. A survey of methods for scaling up inductive algorithms. *Data Min. Knowl. Discov.*, pages 131–169.

J. R. Quinlan. Induction of decision trees. *Mach. Learn.*, 1(1):81–106, March 1986.

J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

Lior Rokach and Oded Maimon. *Data Mining with Decision Trees: Theroy and Applications*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 2008.

Schigehiko Schamoni. Ranking with boosted decision trees, 2012. URL `http://www.ccs.neu.edu/home/vip/teach/MLcourse/4_boosting/materials/Schamoni_boosteddecisiontrees.pdf`.

Robert E. Schapire and Yoav Freund. *Boosting: Foundations and Algorithms*. The MIT Press, 2012.

John C. Shafer, Rakesh Agrawal, and Manish Mehta. SPRINT: A scalable parallel classifier for data mining. In *VLDB*, pages 544–555, 1996.

Apache Spark. Decision trees - RDD-based API, 2012. URL `http://spark.apache.org/docs/latest/mllib-decision-tree.html`.

Anurag Srivastava, Eui-Hong Han, Vipin Kumar, and Vineet Singh. Parallel formulations of decision-tree classification algorithms. *Data Min. Knowl. Discov.*, pages 237–261, 1998.

Stephen Tyree, Kilian Q. Weinberger, Kunal Agrawal, and Jennifer Paykin. Parallel boosted regression trees for web search ranking. In *WWW*, pages 387–396, 2011.

Wikipedia. Discounted cumulative gain. URL https://en.wikipedia.org/wiki/Discounted_cumulative_gain.

Jerry Ye, Jyh-Herng Chow, Jiang Chen, and Zhaohui Zheng. Stochastic gradient boosted distributed decision trees. In *CIKM*, pages 2061–2064, 2009.

Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, pages 15–28, 2012.

Matthew D. Zeiler. ADADELTA: an adaptive learning rate method. *CoRR*, 2012. URL http://arxiv.org/abs/1212.5701.

Zhaohui Zheng, Hongyuan Zha, Tong Zhang, Olivier Chapelle, Keke Chen, and Gordon Sun. A general boosting method and its application to learning ranking functions for web search neur. In *NIPS*, pages 1697–1704, 2008.