Yggdrasil: An Optimized System for Training Deep Decision Trees at Scale

Firas Abuzaid¹, Joseph Bradley², Feynman Liang³, Andrew Feng⁴, Lee Yang⁴, Matei Zaharia¹, Ameet Talwalkar⁵
¹MIT CSAIL, ²Databricks, ³University of Cambridge, ⁴Yahoo, ⁵UCLA

Abstract

Deep distributed decision trees and tree ensembles have grown in importance due to the increasing need to model high-dimensional data. However, PLANET, the standard distributed tree learning algorithm implemented in systems such as XGBOOST and Spark MLlib, scales poorly as data dimensionality and tree depths grow. We present YGGDRASIL, a new distributed tree learning method that outperforms existing methods by up to 24×. Unlike PLANET, YGGDRASIL is based on vertical partitioning of the data (i.e., partitioning by feature), along with a set of optimized data structures that reduce both the CPU and communication cost of training. In particular, YGGDRASIL (1) trains directly on compressed data for compressible features and labels; (2) introduces efficient data structures for training on uncompressed data; and (3) minimizes communication between nodes by using sparse bitvectors. YGGDRASIL is already in production use at a large Web company, and we show that it achieves order-of-magnitude speedups on high-dimensional datasets.

1 Introduction

Decision tree based methods, such as random forests and gradient-boosted trees, have a rich and successful history in the machine learning literature. They remain some of the most widely-used models for both regression and classification tasks, and have proven to be practically advantageous for several reasons: they are arbitrarily expressive, can naturally handle categorical features, and are robust to a wide range of hyperparameter settings [3].

As datasets have grown in scale, there is an increasing need for distributed algorithms to train decision trees. Google’s PLANET framework [10] has been the de facto approach for distributed tree learning, with several popular open source implementations, including Apache Mahout, Spark MLlib, and XGBOOST [1] [9] [6]. PLANET partitions the training instances across machines and parallelizes the computation of split points and stopping criteria over them, thus effectively leveraging a large cluster.

Unfortunately, while PLANET works well for shallow trees and small numbers of features, it has a high communication cost when tree depths and data dimensionality grow. PLANET’s communication cost is linear in the number of features $p$, and is proportional to $2^D$, where $D$ is the tree depth. These deficiencies for large $p$ and large $D$ are problematic, as datasets have become increasingly high-dimensional and complex, often requiring high-capacity models (e.g., deep trees) to achieve good predictive accuracy, as demonstrated by several studies [11] [2] [7].

We present YGGDRASIL, a new distributed tree learning system that scales well to high-dimensional data and deep trees. Unlike PLANET, YGGDRASIL is based on vertical partitioning of the data [4]: it assigns a subset of the features to each worker machine, and asks it to compute an optimal split for each of its features. These candidate splits are then sent to a master, which selects the best one. On top of the basic idea of vertical partitioning, YGGDRASIL introduces three novel optimizations:
• **Training on compressed data without decompression:** YGGDRASIL compresses features via run-length encoding and encodes labels using dictionary compression. We design a novel split-finding scheme that trains directly on compressed data for compressible features, which reduces runtime by up to 20%.

• **Efficient training on uncompressed data:** YGGDRASIL’s data structures let each worker implicitly store the split history of the tree without introducing any memory overheads. Each worker requires only a sequential scan over the data to perform greedy split-finding across all leaf nodes in the tree, and only one set of sufficient statistics is kept in memory at a time.

• **Minimal communication between nodes:** YGGDRASIL uses sparse bit vectors to reduce inter-machine communication costs during training.

Together, these optimizations yield an algorithm that is asymptotically less expensive than PLANET on high-dimensional data and deep trees: YGGDRASIL’s communication cost is \(O(2^D + Dn)\), in contrast to \(O(2^Dp)\) for PLANET-based methods, and its data structure optimizations yield up to 2× savings in memory and 40% saving in time over a naive implementation of vertical partitioning. These optimizations enable YGGDRASIL to scale to handle thousands of features and tree depths up to 20. Specifically, on tree depths greater than 10, YGGDRASIL outperforms MLLIB and XGBOOST by up to 6× on the MNIST 8M dataset, and up to 24× on a dataset with 2 million training examples and 3500 features modeled after the production workload at Yahoo.

**Notation** We define \(n\) and \(p\) as the number of instances and features in the training set, \(D\) as the maximum depth of the tree, \(B\) as the number of histogram buckets to use in PLANET, \(k\) as the number of workers in the cluster, and denote \(W_j\) as the \(j\)th worker.

## 2 YGGDRASIL: Vertical Partitioning

We propose an alternative algorithm to address the communication shortcomings of the horizontally partitioned approach. This algorithm assumes that the data is stored in a distributed fashion, but with the data partitioned by *feature* instead of by *instance*. In other words, each of the \(k\) worker machines stores all feature values for \(\lceil \frac{F}{k} \rceil\) of the features, as well the labels for all instances. This organizational strategy has two crucial benefits: (1) each worker can locally compute the node purity for a subset of the split candidates, which significantly reduces the communication bottleneck; and (2) we can efficiently consider all possible \(B = n - 1\) splits.

We can derive an expression for the splitting criterion \(\text{Split}(\cdot)\) for node \(i\) as

\[
\text{Split}(i) = \arg \max_{s \in \mathcal{S}} \sum_{x \in I} g(x, s) \quad \text{for functions } f \text{ and } g \text{ where } f : \mathbb{R}^c \rightarrow \mathbb{R}, \quad g : \mathbb{R}^c \times \mathbb{N} \rightarrow \mathbb{R}^c,
\]

\(c \in O(1)\), and \(\mathcal{S}\) is the set of features stored on the \(j\)th worker:

\[
\text{Split}(i) = \arg \max_{j} f_j \quad \text{where} \quad f_j = \arg \max_{s \in \mathcal{J}} f \left( \sum_{x \in I} g(x, s) \right)
\]

(1)

Intuitively, each worker identifies its top split candidate among its set of features, and the master then chooses the best split candidate among these top \(k\) candidates.

As with horizontal partitioning, the computation is linear in \(n, p, D\), and is easily parallelizable. However, the communication profile is quite different, with two major sources of communication. For each node, each worker must communicate one tuple of size \(c\) in step 2, resulting in \(2^Dkc\) communication for all nodes. Additionally, when training each level of the tree, \(n\) bits must be communicated in step 5 to indicate the splitting behavior for each training point. Hence, the overall communication is \(O(2^Dk + Dnk)\). In contrast to the \(O(2^DkpB)\) communication cost of horizontal partitioning, vertical partitioning has no dependence on \(p\), and, for large \(n\), the \(O(Dnk)\) term will likely be the bottleneck.

Thus, there exists a set of tradeoffs between horizontal and vertical partitioning across different regimes of \(n, p, D\), as illustrated in Figure 1. The overall trend is clear: for large \(p\) and \(D\), vertical partitioning can drastically reduce communication.

### 2.1 Algorithm

The YGGDRASIL algorithm works as follows: at iteration \(t\), we compute the optimal splits for all nodes on the \(t\)th level of the tree via two round trips of communication between the master and the
workers. Like PLANET, all splits for a single depth \( t \) are computed at once. For each node \( i \) at depth \( t \), the following steps are performed:

**ComputeBestSplit(\( i \)):**
- The \( j \)th worker locally computes \( f_j \) from **Equation 1** and sends this to the master.
- The master selects \( s^* = \text{Split}(i) \). Let \( f_j^* \) denote the optimal feature selected for \( s^* \), and let \( W_j^* \) be the worker containing this optimal feature: \( f_j^* \in W_j^* \).

**bitVector = CollectBitVector(\( W_j^* \))**:
- The master requests a bitvector from \( W_j^* \) in order to determine which child node (either left or right) each training point \( x \in I \) should be assigned to.

**BroadcastSplitInfo(bitVector)**:
- The master then broadcasts the bitvector to all \( k \) workers. Each worker then updates its internal state to prepare for the next iteration of training.

### 2.2 Optimizations

As we previously showed, vertical partitioning leads to asymptotically lower communication costs as \( p \) and \( D \) increase. However, this asymptotic behavior does not necessarily translate to more efficient tree learning; on the contrary, a naive implementation may easily lead to high CPU and memory overheads, communication overhead, and poor utilization of the CPU cache. In YGGDRASIL, we introduce three novel optimizations for vertically partitioned tree learning that significantly improve its scalability, memory usage and performance.

#### 2.2.1 Sparse Bitvectors for Reduced Communication Overhead

Once the master has found the optimal split \( s^* \) for each leaf node \( i \) in the tree, each worker must then update its local features to reflect that the instances have been divided into new child nodes. To accomplish this while minimizing communication, the workers and master communicate using bitvectors. Specifically, after finding the optimal split, the master requests from worker \( W_j^* \) a corresponding bitvector for \( s^* \); this bitvector encodes the partitioning of instances between the two children of \( i \). Once the master has collected all optimal splits for all leaf nodes, it broadcasts the bitvectors out to all workers. This means that (assuming a fully balanced tree), for every depth \( t \) during training, \( 2^t \) bitvectors – for a total of \( n \) bits – are sent from the \( k \) workers.

Additionally, the \( n \) bits are encoded in a sparse format [5], which offers much better compression via packed arrays than a naive bitvector. This sparse encoding is particularly useful for imbalanced trees: rather than allocate memory to encode a potential split for all nodes at depth \( t \), we only allocate memory for the nodes in which an optimal split was found. By taking advantage of sparsity, we can send the \( n \) bits between the master and the workers at only a fraction of the cost.
2.2.2 Training on Compressed Data without Decompression

In addition to its more favorable communication cost for large $p$ and $D$, YGGDRASIL’s vertical partitioning strategy presents a unique optimization opportunity: the ability to efficiently compress data by feature. Furthermore, because the feature values must be in sorted order to perform greedy split-finding, we can use this to our advantage to perform lossless compression without sacrificing recoverability. This leads to a clear optimization: feature compression via run-length encoding (RLE), an idea that has been explored extensively in column-store databases [8, 12]. In addition to the obvious in-memory savings, this technique also impacts the runtime performance of split-finding, since the vast majority of feature values are now able to reside in the L3 cache. To the best of our knowledge, YGGDRASIL is the first system to apply this optimization to decision tree learning.

Many features compress well using RLE: sparse features, continuous features with few distinct values, and categorical features with low arity. However, to train directly on compressed data without decompressing, we must maintain the feature in sorted order throughout the duration of training, a prerequisite for RLE. Therefore, to compute all splits for a given depth $t$, we introduce a data structure to record the most recent splits at depth $t-1$. Specifically, we create a mapping between each feature value and the node $i$ at depth $t$ that it is currently assigned to.

At the end of an iteration of training, each worker updates this data structure by applying the bitvector it receives from the master, which requires a single sequential scan over the data. All random accesses are confined to the labels, which we also encode (when feasible) using dictionary compression. This gives us much better cache density during split-finding: all random accesses no longer touch DRAM and instead read from the last-level cache.

To minimize the number of additional passes, we compute the optimal split across all leaf nodes as we iterate over a given feature. This means that each feature requires only two sequential scans over the data for each iteration of training: one to update the value-node mapping, and one to compute the entire set of optimal splits for iteration $t+1$. However, as a tradeoff, we must maintain the sufficient statistics for all splits in memory as we scan over the feature. For categorical features (especially those with high arity), this cost in memory overhead proves to be too exorbitant, and the runtime performance suffers despite obtaining excellent compression. For sparse continuous features, however, the improvements are significant: on MNIST 8M, we achieve $2\times$ compression (including the auxiliary data structure) and obtain a 20% reduction in runtime.

2.2.3 Efficient Training on Uncompressed Data

For features that aren’t highly compressible, YGGDRASIL uses a different scheme, one that, in contrast, does not use any auxiliary data structures to keep track of the split history. Since the features no longer need to stay sorted in perpetuity, YGGDRASIL implicitly encodes the split partitions by recursively dividing its features into sub-arrays – each feature value is assigned to a sub-array based on the bit assigned to it and its previous sub-array assignment. Because the feature is initially sorted, a sequential scan over the sub-arrays maintains the sorted-order invariant, and we construct the sub-arrays for the next iteration of training in $O(n)$ time, requiring only a single pass over the feature. (For additional clarification, see Figure 5 in the Appendix.) By using this implicit representation of the split history, we’re left only with the feature values and label indices stored in memory. Therefore, the memory load does not increase during training for uncompressed features – it remains constant at $2\times$. This scheme yields another additional benefit: when computing the next iteration of splits for depth $t+1$, YGGDRASIL only needs to maintain the sufficient statistics for one node at a time, rather than for all leaf nodes. Furthermore, YGGDRASIL still only requires a single sequential scan through the entire feature to compute all splits for the entire depth $t+1$. This means that, as was the case for training on compressed features, every iteration requires only two sequential scans over each feature, and all random accesses are again confined to the dictionary-compressed labels, which maximizes our cache utilization and memory bandwidth as before. Finally, for imbalanced trees, we can skip entire sub-arrays that no longer need to be split, which saves additional time as trees grow deeper.

3 Evaluation

We developed YGGDRASIL on top of Spark 1.6.0 with an API compatible with MLlib. Our implementation is 1385 lines of code, excluding comments and whitespace. Our implementation
Our experimental results show that, for large $p$ and $D$, YGGDRASIL outperforms PLANET by an order of magnitude, corroborating our analysis in Section 2.

### 3.1 Experimental Setup

We benchmarked YGGDRASIL against two implementations of PLANET: Spark MLlib v1.6.0, and XGBOOST4J-Spark v0.47. These two implementations are not exactly identical to the algorithm presented by Panda et al. In particular, the original PLANET algorithm has two separate subroutines: one for distributed training, and one for “local” training. By default, PLANET executes the horizontally partitioned algorithm detailed in [10] using on-disk data; however, if the instances assigned to a given node in the tree fit in the memory of a single worker, then PLANET moves all the data for that node to one worker and switches to in-memory training on that worker. In contrast, MLlib loads all the data into distributed memory across the cluster at the beginning of the computation and executes all training passes in memory. XGBOOST extends PLANET with several additional optimizations; see [6] for more details.

We evaluated all of our experiments on 16 r3.2xlarge machines from Amazon EC2. Each machine was equipped with an Intel Xeon E5-2670 v2 CPU, 61 GB of memory, and 1 Gigabit Ethernet connectivity. Prior to our experiments, we appropriately tuned Spark’s memory configurations (e.g., fraction of heap memory used for storage, number of partitions, etc.) for optimal performance. All results reported are averaged over five trials.

### 3.2 Large-scale experiments

To examine the performance of YGGDRASIL and PLANET, we trained a decision tree on two large-scale datasets: the MNIST 8 million dataset, and another modeled after a private dataset provided by Yahoo. Table 1 summarizes the parameters of these datasets.

<table>
<thead>
<tr>
<th>Dataset</th>
<th># instances</th>
<th># features</th>
<th>Size</th>
<th>Task</th>
</tr>
</thead>
<tbody>
<tr>
<td>MNIST 8M</td>
<td>$8.1 \times 10^8$</td>
<td>784</td>
<td>18.2 GiB</td>
<td>classification</td>
</tr>
<tr>
<td>Yahoo 2M</td>
<td>$2 \times 10^6$</td>
<td>3500</td>
<td>52.2 GiB</td>
<td>regression</td>
</tr>
</tbody>
</table>

Table 1: Parameters of the datasets for our experiments

Figure 2 shows the training time across various tree depths for the MNIST 8M and Yahoo 2M datasets. For both datasets, we carefully tuned XGBOOST to run on the maximum number of threads and the optimal number of partitions. Despite this, XGBOOST was unable to train trees deeper than $D = 13$ without crashing due to OutOfMemory exceptions. While Spark MLlib’s implementation of PLANET is marginally faster for shallow trees, its runtime increases exponentially as $D$ increases. YGGDRASIL, on the other hand, scales well up to $D = 20$, at which it runs up to $6\times$ faster. For the Yahoo 2M dataset, YGGDRASIL’s speed-up is even greater because of the higher number of features.

---

1. Yggdrasil has been published as a Spark package at the following URL: [https://spark-packages.org/package/fabuzaid21/yggdrasil](https://spark-packages.org/package/fabuzaid21/yggdrasil)
$p$ – recall that the communication cost for PLANET is proportional to $2^D$ and $p$. Thus, for $D = 18$, YGGDRASIL is up to $24 \times$ faster than Spark MLlib.

### 3.3 Study of Individual Optimizations

To highlight the impact of the optimizations in Section 2.2, we measure the impact of each optimization and see its effect on the runtime for YGGDRASIL. To fully evaluate our optimizations – including feature compression – we chose MNIST 8M, whose features are all sparse, for this study. The results of this experiment are shown in Figure 3; we see that the total improvement from the naive baseline to the fully optimized algorithm is a 40% reduction in runtime. As discussed previously, the use of sparse bitvectors reduces the communication overhead between the master and the workers, thus giving us a modest speedup. Moreover, encoding the labels and compressing the features via run-length encoding both yield a 20% improvement, respectively. As discussed before, these speedups are due to improved cache utilization: encoding the labels via dictionary compression reduces their size in memory by $8 \times$; as a result, the labels entirely fit in the last-level cache. The feature values also fit in the cache after applying RLE, and we gain $2 \times$ in memory overhead once we factor in the necessary auxiliary data structures. Overall, we find that our optimizations for training on compressed data have a substantial impact on the runtime performance during training.

### 4 Related Work

**Distributed Tree Learning.** The most widely used distributed tree learning method is PLANET ([10]), which is also implemented in open source libraries such as Apache Mahout ([1]) and MLlib ([9]). As shown in Figure 1, PLANET works well for shallow trees and small numbers of features, but its cost grows quickly with tree depth and is proportional to the number of features and the number of bins used for discretization. This makes it suboptimal for some large-scale tree learning problems.

XGBOOST ([6]) uses a similar partitioning scheme to PLANET, in that it distributes data across the workers by row, but it uses a compressed, sorted columnar format inside each “block” of data. Its communication cost is therefore similar to PLANET, but its memory consumption is smaller. XGBOOST is optimized for gradient-boosted trees, in which case each tree is relatively shallow. Therefore it does not perform as well as YGGDRASIL on deeper trees, such as those needed for random forests, as we show in our evaluation. XGBOOST also lacks some of the processing optimizations in YGGDRASIL discussed in Section 2.2, such as label encoding to maximize cache density and training directly on run-length encoded features without decompressing.

### 5 Conclusion

Decision trees and tree ensembles are an important class of models, but the current distributed algorithms for training them are optimized for small numbers of features and shallow trees. We have presented YGGDRASIL, a new distributed tree learning system optimized for deep trees and thousands of features. Through vertical partitioning of the data and a set of data structure and algorithmic optimizations, YGGDRASIL outperforms existing tree learning systems by up to $24 \times$, while simultaneously eliminating the need to approximate data through binning. YGGDRASIL is easily implementable on parallel engines like MapReduce and Spark.
References


A Appendix

A.1 Schematic diagram of PLANET vs YGGDRASIL

\[ \bar{X} = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} \in \mathbb{R}^{n \times p} \]

\[ \bar{X} = \begin{bmatrix} c_1 \\ \vdots \\ c_p \end{bmatrix} \in \mathbb{R}^{n \times p} \]

Figure 4: Schematic overview of how PLANET (left) and YGGDRASIL (right) distribute the computation for finding the optimal split candidate \( s \) for a single node in the tree.

A.2 Uncompressed Training in YGGDRASIL

\[ c_i = \text{bitVector} = 100101 \]

\[ \text{sort by value} \]

\[ \text{split found, sort by bitvector} \]

\[ \text{Entire Cluster} \]

\[ \text{Single Worker} \]

Figure 5: Overview of one iteration of uncompressed training in YGGDRASIL. Left side: Root node \( i_0 \) is split into nodes \( i_1 \) and \( i_2 \); the split is encoded by a bitvector. Right side: Prior to training, the feature \( c_i \) is sorted to optimize split-finding. Once a split has been found, \( c_i \) is re-sorted into two sub-arrays: the 1st, 4th, and last values (the “on” bits) are sorted into \( i_1 \)’s sub-array, and the “off” bits are sorted into \( i_2 \)’s sub-array. Each sub-array is in sorted order for the next iteration of training.