Distributed Task-Based Training of Tree Models

Da Yan[*1], Md Mashiur Rahma Chowdhury[*2], Guimu Guo[*3], Jalal Kahlil[*4], Zhe Jiang[†5], Sushil Prasad[‡6]

[*] *Department of Computer Science, The University of Alabama at Birmingham*
{[1]yanda, [2]mashiur, [3]guimuguo, [4]jalalk}@uab.edu
[†] *Department of Computer & Information Science & Engineering, University of Florida*
[5]zhe.jiang@ufl.edu
[‡] *Department of Computer Science, The University of Texas at San Antonio*
[6]sushil.prasad@utsa.edu

***Abstract*—Decision trees and tree ensembles are popular supervised learning models on tabular data. Two recent research trends on tree models stand out: (1) bigger and deeper models with many trees, and (2) scalable distributed training frameworks. However, existing implementations on distributed systems are *IO-bound* leaving CPU cores underutilized. They also only find best node-splitting conditions *approximately* due to row-based data partitioning scheme. In this paper, we target the exact training of tree models by effectively utilizing the available CPU cores. The resulting system called *TreeServer* adopts a *column-based* data partitioning scheme to minimize communication, and a *node-centric task-based* engine to fully explore the CPU parallelism. Experiments show that TreeServer is up to 10× faster than models in Spark MLlib. We also showcase TreeServer's high training throughput by using it to build big "deep forest" models.**

I. Introduction

Decision tree is a supervised model trained over a data table to predict the value of a target attribute Y for an entity based on the observed values of its other attributes $A_1, A_2, \ldots, A_m$. For example, given a customer data table in Fig. 1(a), Fig. 1(b) shows a decision tree to predict whether someone may default on a credit card payment. To reach a decision (Y-value at a leaf), a list of questions on attributes like "Age" and "Income" are gone through starting from the root.

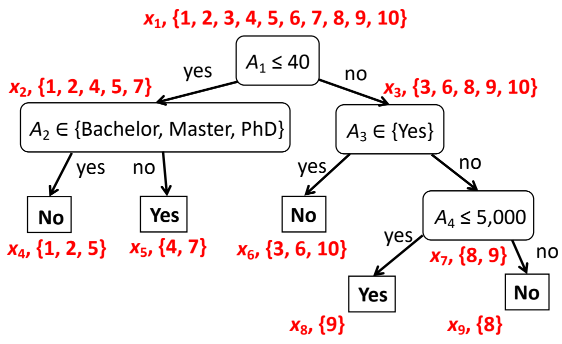
Recently, deeper ensemble models with many trees like deep forests [34] and mGBDTs [19] were shown to outperform deep neural networks in a number of tasks. Moreover, the distributed tree training algorithm, Google's PLANET [27], has been implemented as a standard model in distributed data science libraries such as Mahout [26] and Spark MLlib [24].

However, since PLANET was originally proposed for MapReduce that keeps data on HDFS as row blocks, while nodes in tree models are split by conditions on attributes (i.e., columns), PLANET cannot support efficient computation of the best node-splitting conditions. Note that since the values of each individual column are stored in a distributed manner, excessive communication is necessary to find the best node-splitting conditions. To mitigate this issue, most existing implementations compute approximate node-splitting conditions, such as in Spark MLlib [24] and XGBoost [17].

Note that a Spark-based system such as MLlib cannot fundamentally address this problem, since RDDs are distributed

	A_1: Age	A_2: Highest Education	A_3: Home Owner	A_4: Income	Y: Default
1	24	Bachelor	No	5,000	No
2	28	Master	Yes	7,500	No
3	44	Bachelor	Yes	5,500	No
4	32	Secondary	Yes	6,000	Yes
5	36	PhD	No	10,000	No
6	48	Bachelor	Yes	6,500	No
7	37	Secondary	No	3,000	Yes
8	42	Bachelor	No	6,000	No
9	54	Secondary	No	4,000	Yes
10	47	PhD	Yes	8,000	No

(a) Data Table D



(b) Decision Tree Δ_{x1}

Fig. 1. Data Table and Tree Notations

in-memory row blocks. Even though Spark is switching to a "DataFrame" data organization to support column abstractions, the DataFrames are still row-based and actually immutable. New columns can be computed from DataFrames but they are still stored in a distributed manner, and a machine still cannot access an entire column without communication.

Another efficiency issue with PLANET is that, it adopts a top-down approach for node construction. Due to row-based data partitioning, upper-level node processing is IO-bound, making CPU utilization low initially during tree construction.

Recently, T-thinker [30] has been proposed as a parallel computing paradigm to overcome the problem of CPU under-utilization in existing data-intensive Big Data systems, which were found to have a comparable or lower throughput than a single-threaded program [20], despite the use of aggregate IO throughput from many machines. T-thinker has seen success in graph mining problems [21], [22], [28], [29], [31]–[33].

T-thinker targets divide-and-conquer problems where a computing task over a big dataset can be recursively divided into independent tasks over smaller data subsets for parallel computation. Since the communication cost of collecting data for a task is linear to the data size, while computing workloads grow quickly with the data size, the latter cost can surpass the former as long as the data of a task is not too small. If a task is waiting for data, T-thinker suspends it to release a CPU core for use by other tasks, and the task will be timely resumed when its requested data are all received. This allows communication to be overlapped with computation to keep CPU cores busy with the actual computation.

Following the T-thinker paradigm, we propose *TreeServer*, a distributed system for training tree models where each task corresponds to the construction of a subtree rooted at a node x, denoted by Δ_x; this task can be divided into subtasks that construct subtrees rooted at x's child nodes. If the set of training data that fall into x is small enough for processing by a machine, a task pulls these data to build the entire Δ_x locally without additional communication to keep a core busy.

We identify three challenges in realizing the above idea: (1) how to schedule such CPU-bound tasks early to utilize the idle CPU cores in contrast to PLANET's top-down level-by-level node processing which delays effective CPU utilization? (2) how to compute the best attribute splitting conditions at each node exactly without excessive communication as experienced by current systems? (3) since we adopt a master-workers architecture where the master manages tree construction tasks, how to effectively direct the data-pulling requests from workers to different other workers without master involvement, so that data communication would not overwhelm any machine?

TreeServer provides elegant solutions to all the above 3 challenges. The main contributions of this paper are as follows:

- TreeServer partitions tabular data among machines by columns, which allows each attribute to be checked for node splitting without excessive communication.
- TreeServer schedules node-centric tasks in a hybrid scheme combining breadth-first with depth-first traversal, to schedule CPU-bound subtree construction tasks early.
- TreeServer keeps the data rows associated with different tree nodes in different machines to avoid a single point of communication-bottleneck for transmitting the subset of records requested by child-node tasks.
- Sound algorithms are developed to realize the above designs, plus a cost model for load balancing.

Besides the above desirable designs, TreeServer supports all types of attributes and handles missing data and attribute values unseen during training. TreeServer is fully compatible with the Hadoop ecosystem and loads data in parallel from Hadoop Distributed File System (HDFS). TreeServer is an ideal building block for training larger tree ensembles such as deep forests [34] in a Hadoop analytics workflow. Extensive experiments show that TreeServer is up to 10× faster than Spark MLlib, and constructs deep forests efficiently.

The rest of this paper is organized as follows. Section II reviews our notations and the related work. Section III overviews the design and features of TreeServer. Section IV describes the computation workflow of TreeServer threads. Following that, Section V presents our approach to release master from the duty of relaying row indices of nodes to workers to avoid delaying the transmission of task control messages, and Section VI introduces our approach to assign task workloads to worker machines. Finally, Section VII provides a case study of using TreeServer to construct deep forests, Section VIII reports our experiments, and Section IX concludes this paper.

To ensure reproducibility, we have released our code for TreeServer [11] and deep forest on top [13]. We also provide demo videos for TreeServer (short [10] and full [9] versions) and deep forest [12] with detailed steps to repeat experiments.

II. Preliminaries

Notations and Decision Tree Review. We now introduce our notations, and a complete list of notations used in this paper can be found in Appendix A for quick reference.

We consider a data table D with m attributes $\mathcal{A} = \{A_1, A_2, \ldots, A_m\}$ where one of them is the attribute Y to predict. In Fig. 1(a), we have $m = 5$ and $Y = A_5$. We also denote the set of rows that fall into node x by D_x, and denote their IDs by I_x. In Fig. 1(b), $I_{x_2} = \{1, 2, 4, 5, 7\}$. Finally, we denote the subtree rooted at node x by Δ_x.

During training, at each node x, we want to find a **split-condition** to partition the rows of D_x into child nodes such that an impurity score is reduced the most. We only consider binary node splitting since any multi-way splitting can be represented as a series of binary splitting. Each node x splits D_x based on an attribute A_i called the **split-attribute**.

There are 2 cases for the split-condition: (1) if A_i is ordinal, then the split-condition is "$A_i \leq v$", e.g., node x_1 has $A_1 \leq 40$ in Fig. 1(b); (2) if A_i is categorical, and its possible values constitute a set S_i, then the split-condition is "$A_i \in S_\ell$" where $S_\ell \subset S_i$. If a row in D_x has an attribute value of A_i that falls in S_ℓ, it goes to the left child node (right child otherwise). In Fig. 1, Attribute A_2 has 5 possible values $S_i = \{\text{Primary, Secondary, Bachelor, Master, PhD}\}$, and $S_\ell = \{\text{Bachelor, Master, PhD}\}$. Note that if A_i is categorical, there are exponentially many possible split-conditions.

The split-attribute A_i and its split-condition (i.e., v or S_ℓ) are greedily selected from a set of candidate attributes $\mathcal{C} \subseteq \mathcal{A}$ to minimize impurity after shattering D_x into the two child nodes. The quality of node splitting is evaluated quantitatively using an impurity function such as the entropy of Y-labels for classification and variance of Y-values for regression. At each node x, one may randomly sample a subset $\mathcal{C} \subseteq \mathcal{A}$ of attributes, and then greedily select the best attribute $A_i \in \mathcal{C}$ and its split-condition. A special case is *completely random decision tree* [18] where to split each node, only one feature is randomly sampled (i.e., $|\mathcal{C}| = 1$). In other cases, $\mathcal{C}$ may be sampled beforehand and then used throughout the building of a tree, as is in the ensemble model *random forest* where each tree is trained on a randomly sampled attribute subset.

At each tree node x, we select the best split-condition for each individual attribute (column) A_i independently (hence can run in parallel), often just need one pass over the column values of A_i and Y for rows in D_x using well-known methods for decision tree training. The detailed algorithm can be found in Appendix B. The best conditions of different attributes $A_i \in \mathcal{C}$ are then compared to select x's overall best split-condition.

A node x becomes a leaf for reasons such as (1) all rows in D_x have the same Y-label and thus the prediction at the node is unique; (2) $|D_x| \leq \tau_{leaf}$ where τ_{leaf} is a user-defined threshold; (3) the **depth** of node x, denoted by $d(x)$, reaches the maximum allowed threshold d_{max}. Conditions (2) and (3) are to avoid overfitting the training data, and in classification, the label is predicted as the label that most rows in D_x have, or simply a PMF (probability mass function) vector over all labels. For example, in Fig. 1, if we stop at node x_7 as a leaf, since D_{x_7} contains Row 8 with label "No" and 9 with label "Yes", x_7 will output a PMF vector {"Yes": 50%, "No": 50%} or a predicted label "Yes" (or "No" as we have a tie). For regression where Y is numerical, only conditions (2) and (3) applies, and the predicted value is the average Y-value of D_x.

Related Systems. Since MapReduce and Spark's RDD partition input data among machines by rows, to find the best value v for a split-condition "$A_i \leq v$" for a node x, every machine needs to check its own portion of those rows that belong to D_x. Since there can be up to $n = |D_x|$ different A_i-values for rows in D_x, we need to consider up to $(n-1)$ different split-values for v. For each value of v, different machines need to compare the A_i-values of their portion of rows against v, and the resulting statistics need to be globally aggregated to compute the impurity of splitting. This cost is prohibitive since $(n-1)$ statistics transmissions are needed to examine just one attribute A_i for a node x. As a result, existing systems sacrifice accuracy for efficiency by only sampling a small set of splitting values of D_x for examination. For example, PLANET (adopted by MLlib and Mahout) computes approximate equi-depth histograms for each attribute [27], and a single splitting value is considered from every histogram bucket; XGBoost proposes a weighted quantile sketch for a similar purpose [17] but additionally updates sketches for each node x to reflect attribute values only in D_x rather than in the entire table.

PLANET also proposes to construct an entire subtree Δ_x in a single machine when D_x is small enough, but such CPU-bound computation does not happen until towards the very end of tree construction, before which CPU cores remain underutilized. This is because PLANET constructs nodes top-down level by level, where upper-level nodes require inter-machine collaboration to find the best split-conditions (due to the row partitioning scheme of MapReduce upon which PLANET is built). PLANET constructs all nodes in a level by one MapReduce job, so that each row is read exactly once by a mapper task from HDFS to be IO-efficient. However, this design prevents tasks that build entire subtrees in a machine from being scheduled to run earlier to utilize idle CPU cores.

Among other related works, PV-Tree [23] proposes a communication-efficient heuristic to find split-condition ap-

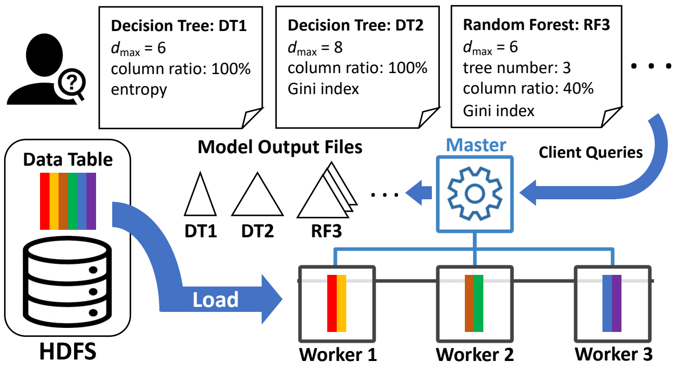


Fig. 2. Master-Workers Architecture of TreeServer

proximately in 3 steps: (1) selecting K attributes with the largest information gain from rows in each local machine, (2) finding $2K$ global attributes among these candidates by majority voting, and (3) collecting histograms of these $2K$ attributes to identify the best attribute and its split-condition. Yggdrasil [15] proposes to significantly reduce the data footprint by compression, and it finds the best split-conditions exactly. Their paper also analytically characterizes the impact of PLANET's approximation to justify the need of exact computation. However, Yggdrasil still adopts a top-down level-by-level node construction order which is IO-bound and does not allow an entire subtree Δ_x to be built in a single machine. Yggdrasil also uses a master to broadcast a bitvector of "row to child-node" assignment to all machines causing a single point of transmission bottleneck. Our TreeServer's algorithm addresses both these problems. We remark that TreeServer can be extended to utilize compression similarly for further speedup, but as different attributes need different compression schemes which require additional user efforts to specify, we do not adopt it to keep our user API data-type transparent.

III. System Overview

TreeServer Architecture. Fig. 2 illustrates how TreeServer works. Specifically, TreeServer adopts a **master-workers** architecture. Users submit their model training jobs only to a master machine, which disassembles each tree model into individual decision trees for training. The master manages the training progress of each decision tree by distributing tasks to worker machines for computation and then collecting their results. The master then assembles the completed trees back into the target tree models to output. For example, in Fig. 2, a user submitted three jobs to train two decision trees DT1 and DT2, and a random forest RF3 with three trees, respectively. The master manages the training of all these decision trees (5 in total), and when completed, reassembles the trees into models DT1, DT2 and RF3 for output. Here, our master-workers architecture basically acts as a server for training individual decision trees (hence the name *TreeServer*).

Recall that TreeServer partitions the columns $A_1, A_2, \ldots, A_m$ of a data table among machines so that a machine holding A_i can compute the split-condition of A_i on its own. The columns are partitioned among the worker machines in a

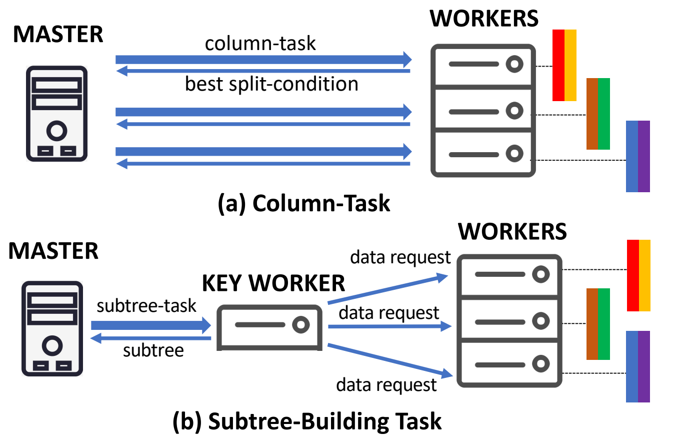


Fig. 3. Illustration of Subtree-tasks and Column-tasks

balanced manner: (1) the target column Y to predict is loaded into (the memory of) every machine since the impurity scores at node x are evaluated based on the Y-values of D_x; (2) each other data column A_i is loaded into k machines. Fig. 2 shows the scenario where $k = 1$, but TreeServer uses $k = 2$ by default since column replicas allow more room for load balancing: the task to compute the split-condition of a column A_i can be assigned to any machine that holds A_i, and we select the one with the minimum current workloads to avoid overloading busy machines; column replicas also support fault tolerance to avoid data loss due to a machine crash.

Task Types. In *TreeServer*, each task t_x is associated with a node x which builds the subtree Δ_x. If D_x is large, t_x is split into subtasks associated with the child nodes of x to allow concurrent processing; while if D_x is small, t_x builds the entire Δ_x. There are two possible task types for a task t_x:

- **Column-task.** To find the best split-condition of a node x, we need to first compute the best split-condition of each attribute $A_i \in \mathcal{C}$. When D_x is large, we let a machine that holds column A_i to compute the best split-condition of A_i for rows of D_x, which is called as a column-task. The best split-conditions computed by concurrent column-tasks are then gathered to find the overall best to split x.
- **Subtree-task.** If D_x is sufficiently small, we let a machine request attribute values of all the rows of D_x from other machines, which are then used to build subtree Δ_x.

Fig. 3 illustrates these two types of tasks. Specifically, **if a task t_x has a large D_x**, the master (1) distributes the column IDs to worker machines as *column-tasks* to find the best column-level split-conditions; it then (2) collects them to determine the overall best condition to split node x (see Fig. 3(a)); finally, the master (3) generates two subtasks t_{x_ℓ} and t_{x_r} for x's left child node x_ℓ and right child node x_r, respectively, to build subtrees Δ_{x_ℓ} and Δ_{x_r}. This essentially partitions the rows of D_x into D_{x_ℓ} and D_{x_r} using the chosen split-condition. **On the other hand, if D_x is small**, the master lets a worker (called as the *key worker*) collect data of D_x from other machines to its local machine to build the subtree Δ_x, which is then sent back to the master to be hooked to node x of the tree under construction (see Fig. 3(b)).

For now, let us assume each task t_x keeps I_x (i.e., the row indices of D_x), but transmitting I_x during task assignment chokes the master's sending channel and is actually avoided in our design as shall be explained later in Section V.

Tree Scheduling. In T-thinker, only the master tracks the decision trees under construction; the other workers are only aware of column-tasks and subtree-tasks assigned by the master.

While our node-centric tasks allow parallel computation within each individual decision tree, in reality, we often need to train many tree models with different hyperparameters for model selection, or train ensemble models with many trees. T-thinker trains all these trees together so that we can have more node-centric tasks to keep CPUs busy. Individual decision trees and those in bagging (e.g., random forest, or a layer in deep forest) can be scheduled in parallel, while in boosting (e.g, gradient boosted trees, or layers in deep forest), sequential dependencies exist where the next layer of trees can only be scheduled for training when all trees in the previous layer is fully constructed. The master keeps track of the tree dependencies, and only considers a tree as a candidate for task scheduling if its prerequisite trees have all been constructed.

The master also controls the pace of tree construction when scheduling the tasks for eligible tree candidates: only a pool of n_{pool} trees are being actively constructed at any time. This keeps the memory usage bounded (as subtree-tasks gather D_x and thus consume memory) and prevents communication overload. When some trees are fully built and thus outputted to release the occupied memory, new eligible tree candidates can then have their tasks scheduled to use the available memory.

Note that no matter how many trees are specified to be trained by users (including those of ensemble models), a tree is flushed to disk by the master as soon as it receives the results from the tree's last task and completes the tree construction, which is tracked with the help of a progress table $\mathcal{T}_{prog}$ at the master. See Appendix C for the implementation details.

Task Scheduling. Within each decision tree, upper-level nodes are processed by TreeServer in parallel using multiple machines to generate enough tasks as soon as possible to be assigned to the available CPU cores, while subtree-tasks of lower-level nodes are also scheduled timely to keep the CPU cores busy. This is achieved through a hybrid task scheduling scheme that combines breadth-first with depth-first traversal.

Specifically, TreeServer takes a job parameter τ_D such that if $|D_x| \leq \tau_D$, task t_x is treated as a subtree-task (column-task otherwise). While subtree-tasks are CPU-bound, column-tasks incur IO workloads due to transmitting I_x (so that workers know which rows to check for columns A_i and Y). TreeServer adopts a hybrid task scheduling scheme *combining breadth-first with depth-first node examination* to allow subtree-tasks and column-tasks to run concurrently to overlap computation with communication. When there are enough subtree-tasks, whenever a column-task is waiting for I_x, it is suspended so that its CPU core can be used to compute another task (e.g., a CPU-bound subtree-task). But when a TreeServer job begins, there are not enough subtree-tasks to saturate the available

CPU cores, so it is favorable to expand each tree level by level to generate more nodes x (hence tasks t_x) to increase parallelism opportunities.

Fig. 4 illustrates the order that nodes (and also their tasks) in a tree are processed in our hybrid scheduling scheme, where upper-level nodes are processed in breadth-first order to quickly generate enough tasks for parallelism. Given a user-defined threshold τ_{dfs} ($> \tau_D$), when $|D_x| \leq \tau_{dfs}$, nodes in subtree Δ_x are then processed in depth-first order (see the dashed subtree); moreover, during this depth-first node traversing, if $|D_x| \leq \tau_D$, then the entire Δ_x is built by a subtree-task executed in one thread (see the black subtrees), in which case compute-bound subtree-tasks are timely scheduled.

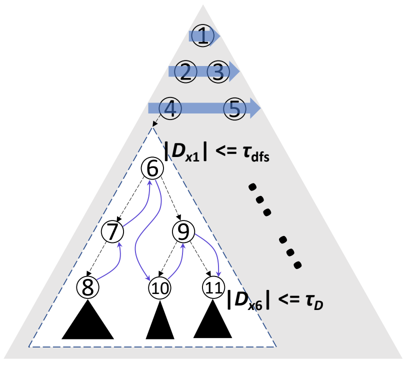


Fig. 4. Task Ordering

Recall that TreeServer dissembles models to train as individual trees which are then trained concurrently, but only at most n_{pool} trees are under construction at any time. This design also helps to schedule subtree-tasks earlier to utilize CPU cores, since otherwise, we have to wait until all (and potentially many) tree "root" tasks have been expanded level by level to where depth-first node traversal begins. Now, when earlier trees in the pool are trained towards the end with subtree-tasks dominating, later comers are still at an early stage with column-tasks dominating, so both kinds of tasks are mixed.

To implement this task scheduler, the master maintains new tasks to be scheduled for computation in a **deque** B_{plan} as shown in Fig. 5 right, which is updated by 2 threads: (1) a main thread θ_{main} that fetches tasks from the head of B_{plan} to be assigned for worker computation, and (2) a receiving thread θ_{recv} that receives the results of column-tasks to find the best split-condition of each node x and that splits the node into subtasks of x's child-nodes to be inserted to B_{plan}. Specifically, when inserting a task t_x: (1) if t_x has $|D_x| > \tau_{dfs}$, then we append it to B_{plan} (like a queue) which essentially performs level-by-level breadth-first node examination, while if $|D_x| \leq \tau_{dfs}$, we add t_x to the head of B_{plan} (like a stack) which essentially performs depth-first examination of the nodes in Δ_x.

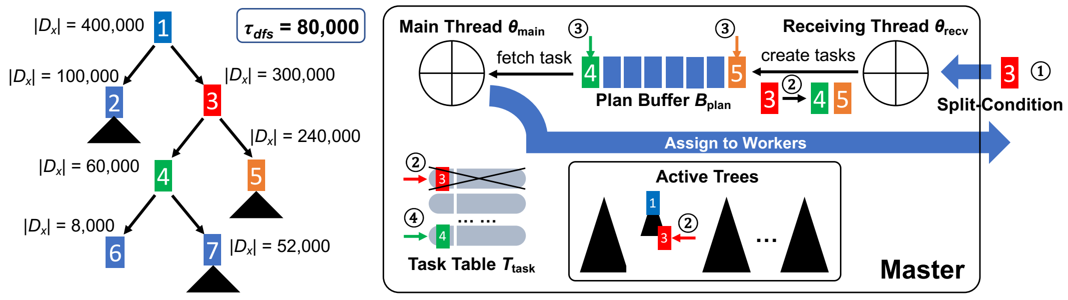


Fig. 5. An Example Decision Tree (Left) & Processing of Nodes 3 and 4 in the Master Machine (Right)

To illustrate, consider Fig. 5 where a decision tree to build is shown on the left with $|D_x|$ of each node x shown beside x. In Fig. 5 right, when the last column-task of node 3 is received by θ_{recv} (see ①), master obtains the overall best split-condition which splits rows of node 3 into child nodes 4 and 5 (see ②). As a result, child tasks for nodes 4 and 5 are created: node 4 is added to the head of B_{plan} since $|D_x| \leq \tau_{dfs}$, while node 5 is added to the tail of B_{plan} since $|D_x| > \tau_{dfs}$ (see ③).

We tuned system parameters τ_D, τ_{dfs} and n_{pool} and found the following default setting that works well consistently on various datasets: $\tau_D = 10,000$, $\tau_{dfs} = 80,000$, $n_{pool} = 200$. The experiments in Section VIII provide more details.

Other TreeServer Features. TreeServer also allows a tree path traversal to stop at any depth to output the predicted label at the current node, which provides additional flexibility in handling missing values and attribute values unseen during training. More details can be found in Appendix D.

IV. System Components

This section introduces the communication channels in T-thinker, and components of the master and workers. We also illustrate the workflow of task creation and processing.

Communication Channels. As Fig. 6 shows, there are 2 types of communication channels: (1) "Task Comm." for the master to send tasks to workers, and for workers to send task results back to the master; (2) "Data Comm." among workers for data requesting and serving: recall that workers pull data from other workers to create D_x when they process subtree-tasks.

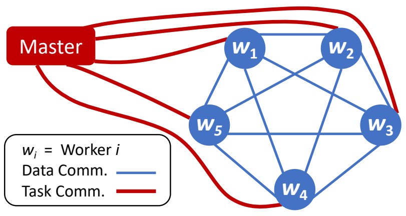


Fig. 6. Communication Channels

Master Components. The master is dedicated to task management and does not compute tasks by itself. Recall from Fig. 5 that the master has two threads: (1) main thread θ_{main} for task assignment and (2) receiving thread θ_{recv} that receives task results and updates the master status.

In T-thinker, tasks are maintained by containers implemented with concurrent data structures for multi-threaded access, such as B_{plan} and T_{task} shown in Fig. 5. The task table T_{task} keeps the status for every task t_x in processing, so that it can be updated properly when θ_{recv} receives the results related to t_x. For example, in Fig. 5, the task for node 3 (denoted by t_3) was added to T_{task} after θ_{main} fetches it from B_{plan} and assigns its column-tasks to the workers. Whenever θ_{recv} receives the split-condition result w.r.t. t_3 from a worker, θ_{recv}

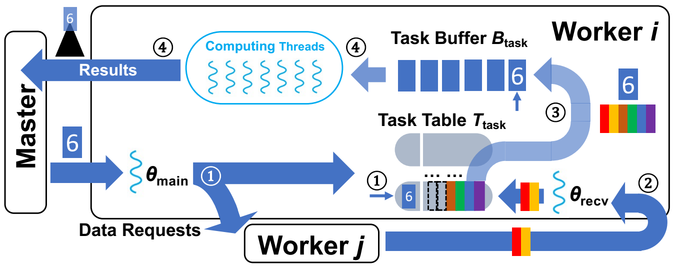


Fig. 7. Components of a Worker

uses it to update the current best split-condition of t_3 in T_{table}, and updates t_3's 'progress bar.' If the last split-condition result w.r.t. t_3 is received by θ_{recv}, the overall best split-condition is found, then node 3 is computed and thus added to its tree under construction, and t_3's entry in T_{table} is freed.

Due to space limit, please refer to Appendix E for more details on task containers and thread workflows in the master.

Worker Components. Workers are the actual workhorses of task computation. Fig. 7 illustrates the components in a worker, where besides (1) a main thread θ_{main} (see ①) for receiving tasks from the master and (2) a receiving thread θ_{recv} (see ②) that receives data from other workers, the worker also maintains (3) a pool of computing threads (aka. **compers**) that fetch tasks from a task buffer B_{task} to compute their results (see ④), which are then sent back to the master.

Here, B_{task} keeps those tasks whose data are locally ready. If a subtree-task t_x is waiting for data from other works to construct D_x, then t_x is kept in the task table T_{task}, and only moved by θ_{recv} to B_{task} when all required data are received.

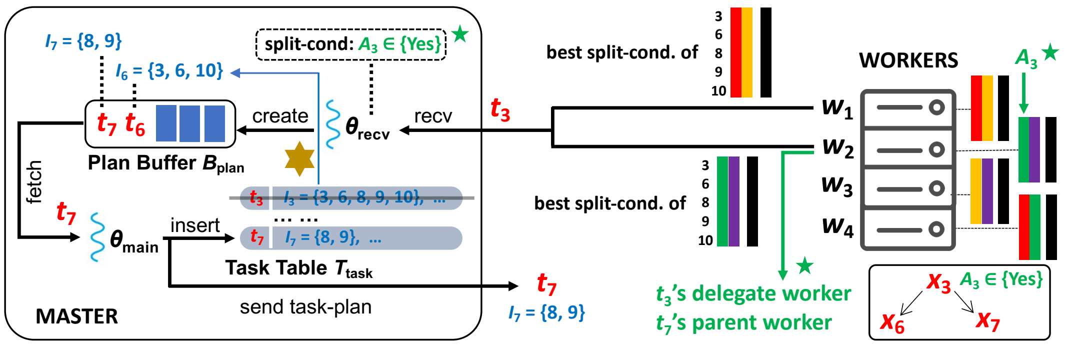
Fig. 7 illustrates the execution flow of subtree-task t_6 for node 6 (recall from Fig. 5 left that its $|D_x| = 8,000 \leq \tau_D = 10,000$) in its key worker (recall from Fig. 3(b) that the key worker of a node x is the worker that collects D_x to build Δ_x). Specifically, when θ_{main} receives the subtree task t_6, it sends requests for the columns of D_x to other workers which loaded these data columns, and meanwhile, it puts t_6 to the task table T_{task} to wait for these columns to be collected (see ①). When θ_{recv} receives the last few columns of D_x from another worker (see ②), D_x is fully collected so θ_{recv} moves t_6 out from T_{task} into B_{task} (see ③) to be fetched by an available comper to construct Δ_x, which is then sent back to the master (see ④) to be added to the corresponding tree.

Due to space limit, please refer to Appendix E for more details on task containers and thread workflows in a worker.

Fault Tolerance. Since a TreeServer program is master-driven, master failure is fatal but can be addressed by keeping a secondary master. If a worker crashes, the master simply reassigns the lost columns to other workers, revokes those tasks t_x that are being processed by the failed worker, and puts t_x back to B_{plan} for reassignment (recall B_{plan} from Fig. 5). More details on fault tolerance can be found in Appendix E.

V. Row Maintenance Without Master Relaying

So far, we have been assuming that each task (or task-plan) object t_x keeps the rows IDs of D_x, i.e., I_x. As Fig. 8(a) illustrates, this applies to task-plans in B_{plan} and task objects in T_{task} in the master, as well as the assigned task-plans sent by the master to workers. However, I_x can be large for task-plans, leading to outbound communication bottleneck at the master. This section will present our technique to avoid letting any task (or task-plan) object t_x keep I_x, so that the communication bottleneck is eliminated. In the sequel, we first define some worker types related to our technique, and then present the motivation and details of our technique.



(a) Task Flow $t_3 \rightarrow t_6$ t_7 in Master (b) Delegate Worker and Parent Worker

Fig. 8. Illustration of Delegate Worker and Parent Worker

Worker Types. Consider Fig. 1 again, where node x_3 is split into nodes x_6 and x_7 using split-condition '$A_3 \in \{\text{Yes}\}$'. Fig. 8 shows this process, where we assume that the column-task t_3 (i.e., t_{x_3} but we abuse the notation for simplicity) generated two plans that were assigned to workers w_1 and w_2, respectively. Also assume that the best split-condition from w_1 as selected from its loaded columns A_1 (red) and A_2 (yellow) has been sent back to the master, as well as the best split-condition from w_2 computed from its columns A_3 (green) and A_4 (purple). As a result, θ_{recv} computes the overall best split-condition '$A_3 \in \{\text{Yes}\}$' which gives the lowest impurity, and creates child tasks t_6 and t_7.

One problem remains here: even though task object $t_3 \in T_{task}$ maintains I_3, the master still does not have access to the attribute values of A_3 to decide which new child node a row in D_x should go to. As a result, another round of communication would be necessary to send the computed split-condition '$A_3 \in \{\text{Yes}\}$' to w_2, which splits I_3 into I_6 and I_7 to be sent back to the master for adding to new child task-plans t_6 and t_7 to be added into B_{plan}.

In general, when a column-task t_x needs to generate child tasks t_{x_ℓ} and t_{x_r} using the best split-condition obtained, assume that the best split-condition is with attribute A_i sent from Worker j, then in order to split I_x into I_{x_ℓ} and I_{x_r}, we have to let Worker j do the splitting since it has the attribute values of A_i to compare with the split-condition. Here, we call Worker j as the **delegate worker** of task t_x. For example, in Fig. 8, w_2 that holds A_3 is the delegate worker of task t_3.

Since the delegate worker splits I_x into I_{x_ℓ} and I_{x_r}, ideally, we would like those workers that process t_{x_ℓ} (resp. t_{x_r}) to obtain I_{x_ℓ} (resp. I_{x_r}) directly from Worker j, rather than ask the master to relay I_{x_ℓ} (resp. I_{x_r}) in its task-plan message that creates repeated communication and outbound communication bottleneck. For example, in Fig. 8, when a task-plan for child task t_7 is assigned to a worker, say w_4, we would like w_4 to directly ask t_3's delegate worker w_2 for I_7, so that the master does not need to send I_7 along with plan t_7 (see Fig. 8(a)).

Here, we delegate the maintenance of I_x for a task t_x directly to its delegate worker, and task (or task-plan) objects t_x in B_{plan} and T_{task} also do not keep I_x. After all, in a constructed tree for prediction, each node x only needs to keep the split-condition rather than I_x, so the master does not have to track I_x. Since different tasks have different delegate workers, communication bottleneck at any machine is avoided.

Let us denote $pa(x)$ as the parent node of x, and denote $sib(x)$ as the sibling node of x. For example, in Fig. 8, $pa(x_7) = x_3$ and $sib(x_7) = x_6$. Then, the worker that handles the requests for I_x is the delegate worker of the "parent" task $t_{pa(x)}$, which has the best split-attribute and thus can split $I_{pa(x)}$ into I_x and $I_{sib(x)}$ using the best split-condition confirmed by master, and which serves both child tasks t_x and $t_{sib(x)}$. We call the delegate worker of parent task $t_{pa(x)}$ simply as the **parent worker** of task t_x.

Technique Overview. Fig. 9 shows the task workflows with our proposed technique, where the name 'parent worker' defined above is adopted. The delegate worker of a column-task t_x now cannot delete t_x's task object from task table T_{task} immediately after the overall best split-condition is notified back by the master. This is because t_x's task object holds I_x and hence I_{x_ℓ} and I_{x_r} (after splitting I_x), both of which will be requested by t_x's child tasks later. After all the workers that process t_x's child tasks have requested I_{x_ℓ} and I_{x_r}, t_x's task object should then be deleted from T_{task}.

Also, since I_x is now tracked by t_x's delegate worker rather than the master, we require a worker that sends back the best split-condition of a column A_i to also send two counters recording $|I_{x_\ell}|$ and $|I_{x_r}|$, so that they can be compared with τ_D to decide the type of child tasks (i.e., subtree-tasks or column-tasks) and whether to insert them into the head or tail of B_{plan}.

Workflow of a Subtree-Task (Fig. 9(a)). For a subtree-task t_x, the master assigns it to a worker for collecting D_x to build the subtree, and we call that worker as the **key worker** of t_x. The plan of a subtree-task indicates that for each attribute $A_i \in \mathcal{C}$, which machine should the key worker ask for attribute values. The key-worker assignment and attribute-to-machine mapping (used by the key worker) are computed by the master.

As Fig. 9(a) shows, the master first sends a subtree plan t_x to its assigned key worker (see the arrow marked with "1"), which creates a task object for t_x and puts it in task table T_{task} to collect D_x. The key worker then (1) requests other workers for attribute values of columns $\mathcal{C}$ for rows of D_x, (2) sends a request to the parent worker to fetch I_x so that the key worker can obtain Y-values for those rows. These 2 kinds of requests are illustrated in Fig. 9(a) by the two arrows marked with "2" (where a thick arrow means multiple messages). When a worker receives the request for attribute values of column-subset $\mathcal{C}' \subseteq \mathcal{C}$ (the request also contains the tracking information of $t_{pa(x)}$ in the parent worker), it then creates a task object of t_x in task table T_{task} and requests I_x from the parent worker (see the arrow marked with "3"). When the worker receives I_x, it then obtains $\mathcal{C}'$ from t_x's task object (the object is then deleted from T_{task}), fetches these rows' attribute values in columns $\mathcal{C}'$ and sends the data back to the key worker (see the arrow marked with "4"). Finally, after the key worker collects the entire D_x, it builds subtree Δ_x and sends it back to master (see the arrow marked with "5").

Workflow of a Column-Task (Fig. 9(b)). The master first sends the column-task plans of t_x to the assigned workers, each of which creates a task object of t_x (attached with the assigned column IDs $\mathcal{C}'$ to find the best split-condition) and puts it in the task table T_{task} to wait for I_x, and then requests the parent worker for I_x. When I_x is received, the worker checks these rows for all columns in $\mathcal{C}'$ to find the best split-

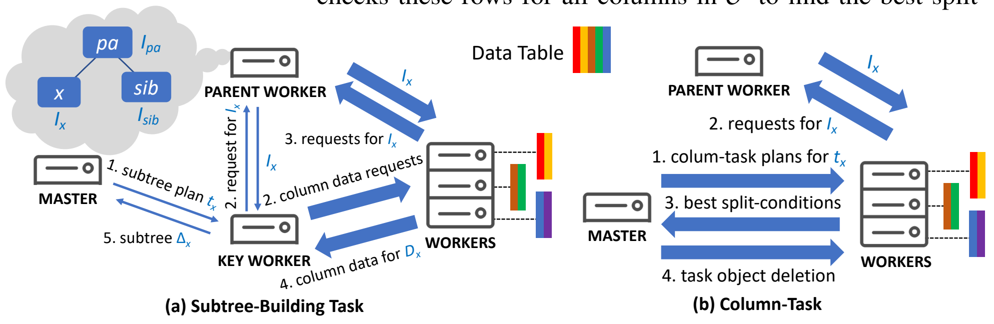


Fig. 9. Task Workflow with Delegate Worker

condition, which is then sent back to the master; by this time, the worker should not delete t_x's task object yet. When the master has received all the responses of t_x and determines that Worker j has the best overall split-condition, it notifies all the other workers to delete their task objects for t_x. The task object of Worker j will be deleted by θ_{recv} right after the last request from t_x's child tasks is served, the progress of which is tracked by the task object and updated after each request for I_{x_ℓ} or I_{x_r} (obtained by splitting I_x using the best split-condition) from a child task.

Assume that t_x is handled by k workers, then only t_x's delegate worker needs to hold I_x for use by child tasks, while the other $(k-1)$ workers can delete their task object t_x immediately after node x is processed, leading to an additional $1/k$ memory cost lingering till all child tasks have requested

I_{x_ℓ} or I_{x_r}. Since t_x is timely deleted by its delegate worker after being used, the additional memory cost is reasonable.

For readability, we omitted minor implementation details, such as skipping communication when the requested data is local, and the task workflow when the key worker of subtree-task t_x finds that D_x should be a leaf (e.g., when all Y-labels are the same so data request to collect D_x is not necessary).

VI. Worker Assignment For Tasks

Since only the master handles task assignment, it has the big picture of plan-to-worker assignment and can keep track of the current worker workload distribution to assign each new plan to those workers that keep the workloads balanced.

Worker ID	Comp	Send	Recv
1	100	70	60
2	60	80	75
3	80	70	90
4	120	60	85
...	... ...		

Fig. 10. Load Matrix $\mathcal{M}_{work}$

Fig. 10 illustrates the matrix $\mathcal{M}_{work}$ that the master uses to track workers' workloads, where each row corresponds to a worker and tracks its estimated current **computation**, **sending** and **receiving** workloads. For example, Row 1 indicates that Worker 1 needs to run 100 units of instructions for task computation, to send 70 units of messages to other machines, and to receive 60 units of messages, where the unit does not matter as long as they are the same for all workers, and the goal is to assign new plans to workers so that the values in every column of $\mathcal{M}_{work}$ become as even (i.e., not drastically different) as possible.

Since (1) there are combinatorially many ways to assign task workloads of a new plan to workers (e.g., key worker and its data-serving workers), and (2) the workload dynamics of tasks that are currently being computed by workers (e.g., different steps in Fig. 9) are difficult to track including the delayed task object release at the delegate workers, it is impractical for the master to make decisions based on the accurate current worker workloads and consider all worker combinations.

To make plan assignment tractable and efficient, we adopt a greedy strategy that (1) treats all workloads of assigned tasks as pending even though they are partially processed, and that (2) makes decisions on each step based only on the dominant cost (i.e., computation or communication). When θ_{main} of the master assigns the workloads of a task-plan t_x to workers, it updates $\mathcal{M}_{work}$ by properly adding workloads; and when θ_{recv} of the master receives the computation result for t_x, it deducts the added workloads (obtained from t_x's task object in task table T_{task} which memorizes the added workloads) from $\mathcal{M}_{work}$ to reflect the completion of t_x. A mutex protects $\mathcal{M}_{work}$ so that only one thread (θ_{main} or θ_{recv}) updates it at a time. We next describe our greedy plan-to-worker assignment.

Assignment of a Subtree-Task. Given a newly-created subtree-task plan t_x, we greedily assign the key worker as the worker with the minimum current **computation workload** in $\mathcal{M}_{work}$ (since t_x is CPU-bound), and add its workload value by $(|I_x| \cdot |\mathcal{C}| \cdot \log |I_x|)$. Here, we assume that all attributes in $\mathcal{C}$ are amenable to one-pass algorithm to find the best split-condition so that the incremental cost of attribute checking for each row is $O(|\mathcal{C}|)$ (as there are $|\mathcal{C}|$ attribute values). Since the nodes of each level of Δ_x partitions the rows of D_x, the overall checking cost for a level is $|I_x| \cdot |\mathcal{C}|$; we also estimate the height of Δ_x as $\log |I_x|$ (assuming balanced Δ_x).

We next assign each column $A_i \in \mathcal{C}$ to a worker so that the key worker of t_x will request it for A_i. We update $\mathcal{M}_{work}$ after each A_i is assigned to a worker, so that the next column can be assigned by taking previous column assignments into consideration. Let us assume that the key worker is Worker i_{key} and the parent worker is Worker i_{pa}. If we assign column A_i to Worker j, then Worker j needs to (1) receive I_x from Worker i_{pa}, and to (2) send the $|I_x|$ attribute values fetched from column A_i to Worker i_{key}. The two transmissions translate into the following 4 updates:

1) $\mathcal{M}_{work}[j][Recv] \leftarrow \mathcal{M}_{work}[j][Recv] + |I_x|$
2) $\mathcal{M}_{work}[i_{pa}][Send] \leftarrow \mathcal{M}_{work}[i_{pa}][Send] + |I_x|$
3) $\mathcal{M}_{work}[j][Send] \leftarrow \mathcal{M}_{work}[j][Send] + |I_x|$
4) $\mathcal{M}_{work}[i_{key}][Recv] \leftarrow \mathcal{M}_{work}[i_{key}][Recv] + |I_x|$

Among them, (3) and (4) are always needed; but to avoid double-counting, (1) and (2) are only needed if $A_i \in \mathcal{C}$ is the first column assigned to Worker j. This is because Worker j will request Worker i_{pa} for I_x only once, for all columns $\mathcal{C}'$ assigned to Worker j (including A_i). To balance the network overheads among workers, Worker j is selected to minimize the maximum of the above 4 updated workload values.

Assignment of a Column-Task. Given a newly-created column-task plan t_x, we determine those workers that examine the columns $\mathcal{C}$ to find the best split-condition. Note that each worker needs to request the parent worker for I_x, and then to examine the attribute values of its assigned columns $\mathcal{C}'$ for these rows. Therefore, if we assign any column A_i to a worker j, we perform updates (1) and (2) exactly like before, and we also add $\mathcal{M}_{work}[j][Comp]$ with the cost of examining A_i (e.g., the cost is $|I_x|$ if a one-pass algorithm is used). Since the column assignment incurs network overheads as a major cost, Worker j is selected to minimize $\max\{\mathcal{M}_{work}[j][Recv], \mathcal{M}_{work}[i_{pa}][Send]\}$ (after their values are updated) to balance communication.

TreeServer also properly skips adding communication workloads in special cases whenever the requested data are local.

VII. A Case Study with Deep Forest

Deep forest [34] is proposed as an alternative to deep neural networks, and it has reported an even higher accuracy. The model consists of many levels of forests each with hundreds of trees, making its training challenging for big data. This section explains how TreeServer can train a deep forest.

Deep Forest Architecture. Fig. 11 shows the structure of a deep forest, consisting of 2 phases: multi-grained scanning (MGS) and cascade forest (CF). MGS uses sliding windows of different sizes ($\mathbf{W}_1$, $\mathbf{W}_2$, $\mathbf{W}_3$) to scan raw features, and the extracted window-sized vectors are used to train forests which are then used to re-represent raw features ($\mathbf{f}_1$, $\mathbf{f}_2$, $\mathbf{f}_3$).

Fig. 12 illustrates the MGS process with a 10×10 window, where the window slides on a raw image data with stride 1.

The extracted feature vectors have dimension $10 \times 10 = 100$, and such vectors from all images are used to train 2 forests. In deep forest, a forest for k-class classification returns a k-dimensional vector computed as the average of the class PMF vectors returned by all its trees. For each image, MGS concatenates the k-D vectors (outputted by both forests) for all window-sized vectors of the image as the inputs, and thus the re-representation can easily have thousands of dimensions.

As Fig. 11 shows, CF Layer 0 takes the re-represented data features from the smallest window as input, and trains forests which are then used to re-represent the data as the concatenation of k-D vectors outputted by the trained forests. In each later level, features outputted from the previous level are concatenated with the re-represented features from a window of particular size during MGS, to train forests to output the next-level re-representations. Prediction at each layer is obtained by averaging the k-D vectors outputted by the last-layer forests, then finding the mostly likely class.

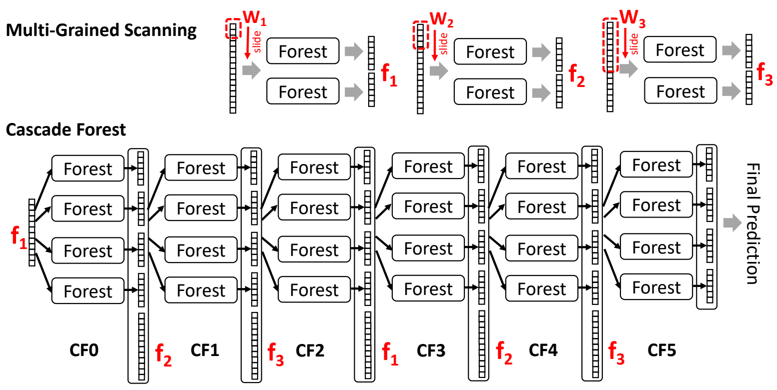


Fig. 11. Overall Procedure of Deep Forest

Implementation in TreeServer. Since each CF layer trains multiple forests (each with many trees), it can be formulated as a TreeServer job. Similarly, training forests from window-sized features in MGS can be formulated as a TreeServer job. We also program 2 parallel operations to work with TreesServer jobs in an entire deep forest construction workflow:

- The first job is to perform window-sliding feature extraction over different images concurrently. For this purpose, image data are partitioned among the threads of all machines for concurrent window-sliding feature extraction.
- Once the forests of a layer have been constructed by TreeServer, they are saved to Hadoop Distributed File System (HDFS). Then, the image data needs to go through these forests again (i.e., the prediction phase) to generate output features (aka. re-representations), which is the task of our second job. Here, we let every machine load all the forests from HDFS, and then conduct tree traversal for its assigned portion of images. Our second job allows different images to be concurrently processed by different threads of different machines.

These two operations partitions input data by rows, while TreeServer model training partitions input data by columns. Their integration calls for a novel data organization on HDFS to be described soon that is friendly to both partition schemes.

Besides random forests, deep forest also supports "completely random decision trees" (aka. extra-trees) as the other forest type. See Appendix F for the details.

Data Organization on HDFS. To support TreeServer's column partitioning scheme, we require users to use our dedicated "put" program (instead of HDFS's) to upload data (e.g., a local CSV file) to HDFS, so that each data column is saved as a file on HDFS that can be loaded by workers in its entirety. Our "put" program is memory-efficient: it uses m HDFS output file streams (one for each column) to append the corresponding attribute values in each data row, as it streams data rows.

A normal tabular data has at most tens to several hundreds of columns, but as we have explained, MGS can easily generate data representations of thousands of dimensions or even more. The many column files would be inefficient for workers to load since HDFS connection time (rather than actual data reads) dominates in our test. Our solution is to group columns to reduce the number of files, as illustrated in Fig. 13 where the input data is organized into 4 column-files each with 50 attributes. Now the column-groups (rather than individual columns) are assigned as the basic unit to the workers in a balanced manner to serve data requests by tasks.

Another necessary change is due to our two additional parallel programs described earlier that partitions data by rows. To support both row- and column-partitioning efficiently, our final data organization adopted is as illustrated in Fig. 13, where each column-group is further partitioned into row-groups, and one file is saved for each row-group. In Fig. 13, now a TreeServer job may load a column-group by reading 4 files in the same column, while our two new parallel programs may load its partition of rows by reading 4 files in the same row. Here, the number of files to read is kept small, and each file contains sufficient data to amortize HDFS connection cost.

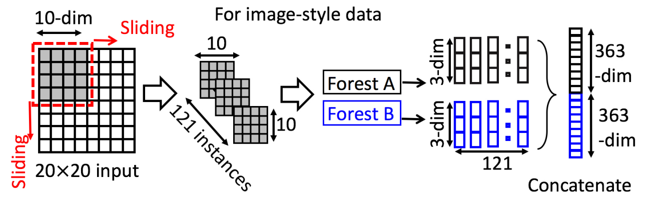


Fig. 12. Image Feature Re-representation

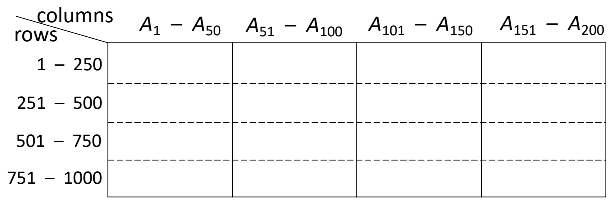


Fig. 13. Data Organization on HDFS

VIII. Experiments

This section evaluates the efficiency of TreeServer, and compares it with the tree models in Spark MLlib [24] and XGBoost [17], the state-of-the-art machine libraries for tree models that support distributed training. Our code is released on GitHub at: https://github.com/yanlab19870714/TreeServer.

All our experiments were conducted on a cluster of 15 machines each with 12 threads (2.67 GHz) and 24 GB RAM. Each experiment was repeated 3 times, with the average results reported. Unless otherwise stated, we run with all 15 machines each with 10 compers, and we set the maximum tree depth $d_{max} = 10$, $\tau_{leaf} = 1$, classification (resp. regression) impurity as Gini index (resp. variance), and $|\mathcal{C}| = |\mathcal{A}|$ for decision trees and $|\mathcal{C}| = \sqrt{|\mathcal{A}|}$ for a tree in random forests.

TABLE I
Datasets

Dataset	#{rows}	#{numerical}	#{categorical}	Problem
Allstate	13,184,290	13	14	regression
Higgs_boson	11,000,000	28	0	
MS_LTRC	723,412	136	1	
c14B	473,134	700	0	
Covtype	581,012	54	0	
Poker	1,025,010	0	11	classification
KDD99	4,898,431	38	3	
SUSY	5,000,000	18	0	
loan_m1	6,372,703			
loan_y1	29,581,722	14	13	
loan_y2	54,468,375			

Datasets. Table I shows the large real datasets we used, each of which either has many rows or many attributes. For all the datasets, we have removed attributes that are not related to prediction, such as ID, date and time, etc. *Allstate* [1], *Higgs_boson* [3], *MS_LTRC* [5], *c14B* [14], *Covtype* [2], *Poker* [6], *KDD99* [4] and *SUSY* [8] are directly usable, among which only *Allstate* is a regression problem and has missing values. TreeServer and XGBoost handle missing values and can directly take *Allstate* as input; while Spark MLlib does not support missing values and so we had to fill missing values with the mean attribute value first. The other dataset *loan* [7] requires preprocessing (see Appendix G for details), and we took the loan data from the latest month, the latest year, and the latest 2 years to obtain 3 datasets of different sizes.

Comparison with MLlib. Spark MLlib uses an argument "maxBins" to indicate the binning size of its attribute value histogram used for finding approximate split-conditions, and we use the default maxBins = 32. Table II(a) (resp. Table II(b)) shows the training time of both TreeServer and MLlib on our datasets for training a decision tree using all columns (resp. training a random forest with 20 trees using $\sqrt{|\mathcal{A}|}$ columns). We can see that TreeServer is consistently many times faster than MLlib (e.g., almost 10× on *MS_LTRC*). Moreover, TreeServer computes the exact split-conditions while MLlib computes split-conditions approximately with attribute binning, and as shown in Tables II(a) & (b), the test accuracy of models trained with TreeServer is slightly higher than that of models from MLlib in the majority of the cases.

Table II(c) compares TreeServer with XGBoost both with 100 trees. Note that TreeServer uses bagging (i.e., random forest) while XGBoost uses boosting. We can see from Table II(c) that XGBoost achieves a higher accuracy on 6 out of the 11 datasets, thanks to its advanced gradient boosting method that considers second-order approximation of the learning objective, but not higher on the other datasets. In contrast, it is much slower than TreeServer. For example, on *C14B* (resp. *Poker*), XGBoost is 56× (resp. 58×) slower than TreeServer. This is because trees in XGBoost have dependencies and have to be trained one after another due to boosting, while trees in TreeServer can be trained together without any dependency.

TABLE II
System Comparison (Accuracy = RMSE for Allstate)

(a) TreeServer v.s. MLlib One Decision Tree

Dataset	TreeServer		MLlib (Parallel)		MLlib (Single Thread)	
	Time (sec)	Accuracy	Time (sec)	Accuracy	Time (sec)	Accuracy
Allstate	56.06	41.46	194.26	41.26	886.98	41.26
Higgs_boson	103.53	70.52%	137.36	69.16%	467.38	69.16%
MS_LTRC	12.21	55.30%	103.45	55.38%	14.60	55.38%
c14B	37.81	67.00%	119.49	49.04%	252.61	49.04%
Covtype	10.00	95.31%	58.39	95.29%	67.14	95.29%
Poker	10.08	53.52%	86.55	53.52%	85.45	53.52%
KDD99	58.58	79.98%	83.90	79.97%	243.00	79.97%
SUSY	38.68	79.13%	90.76	79.10%	156.35	79.10%
loan_m1	52.49	99.63%	322.47	99.50%	524.17	99.50%
loan_y1	284.75	99.64%	551.64	99.64%	1,188.26	99.64%
loan_y2	520.68	99.64%	808.09	99.64%	1,972.12	99.64%

(b) TreeServer v.s. MLlib Random Forest (20 Trees)

Dataset	TreeServer		MLlib (Parallel)		MLlib (Single Thread)	
	Time (sec)	Accuracy	Time (sec)	Accuracy	Time (sec)	Accuracy
Allstate	113.31	41.46	305.86	41.39	1,635.70	41.39
Higgs_boson	135.53	69.89%	259.75	69.69%	978.69	69.69%
MS_LTRC	14.43	55.23%	122.47	55.59%	14.67	55.59%
c14B	29.86	62.00%	125.01	49.59%	277.67	49.59%
Covtype	10.82	91.53%	82.38	91.52%	79.42	91.52%
Poker	10.06	53.71%	174.15	53.95%	175.81	53.95%
KDD99	46.59	80.15%	234.67	80.15%	414.85	80.15%
SUSY	51.81	79.49%	127.59	79.37%	287.29	79.37%
loan_m1	42.44	99.64%	373.25	99.50%	560.28	99.50%
loan_y1	192.73	99.64%	786.94	99.64%	2,856.16	99.64%
loan_y2	367.84	99.64%	1422.20	99.64%	5,008.66	99.64%

(c) TreeServer v.s. XGBoost (100 Trees)

Dataset	TreeServer		XGBoost	
	Time (sec)	Accuracy	Time (sec)	Accuracy
Allstate	381.94	41.46	757.77	38.81
Higgs_boson	429.82	69.85%	2,364.87	70.12%
MS_LTRC	45.38	55.22%	1,585.10	58.83%
c14B	62.33	61.00%	3,492.70	55.85%
Covtype	47.66	92.33%	767.78	93.67%
Poker	23.06	55.95%	1,338.56	56.87%
KDD99	186.03	99.91%	3,892.76	99.93%
SUSY	141.92	79.88%	1,650.33	79.54%
loan_m1	98.52	99.64%	1405.08	99.47%
loan_y1	540.88	99.64%	7,411.35	99.47%
loan_y2	788.66	99.64%	13,390.34	99.47%

Effect of n_{pool}. Recall that to control memory usage, we only allow n_{pool} trees to be built at any time, whose value is 200 by default. To explore the relationship between n_{pool} and a job's peak memory usage (at a machine, averaged over all machines) and running time, we trained a random forest with 20 trees on the various datasets with $n_{pool} = 1, 5, 10$ and 20, respectively. Tables III(a)–(c) show our results on *Allstate*, *Higgs_boson*, and *KDD99* (results on other datasets are similar). We can see that the job running time decreases a lot as n_{pool} increases, though the improvement is less significant as n_{pool} gets larger (e.g., 6× on *Allstate* when $n_{pool} = 20$). This is because our machine is equipped with a Xeon X5650 CPU (6 cores, 12 threads) which is already near saturation given the tasks of

TABLE III
EFFECT OF n_{pool}, τ_{dfs} AND τ_D (TIME UNIT: SECOND; MEM UNIT: GB)

(a) n_{pool} on Allstate

n_{pool}	Time	Memory
1	793.64	5.06
5	216.95	5.20
10	149.98	5.62
20	125.06	5.54

(b) n_{pool} on Higgs_boson

n_{pool}	Time	Memory
1	656.71	3.00
5	220.33	3.41
10	163.13	4.21
20	140.47	4.76

(c) n_{pool} on KDD99

n_{pool}	Time	Memory
1	277.05	1.78
5	76.99	1.97
10	58.07	2.05
20	49.46	2.52

(d) Effect of System Parameter τ_{dfs}

τ_{dfs}	Allstate	Higgs_ boson	KDD99
20,000	121.63	152.19	52.79
50,000	113.85	149.31	50.18
80,000	**108.13***	**134.10***	**46.59***
100,000	112.74	140.63	49.51
150,000	125.58	160.94	53.27

(e) Effect of System Parameter τ_D

τ_D	Allstate	Higgs_ boson	KDD99
2,000	121.72	148.78	50.42
5,000	117.16	148.52	51.91
8,000	116.81	142.52	48.42
10,000	**108.13***	**134.10***	**46.59***
15,000	123.97	148.91	47.96
20,000	125.91	149.05	49.09

TABLE IV
RUNNING TIME V.S. NUMBER OF TREES

(a) MS_LTRC

#{trees}	TreeServer		Spark MLlib	
	Time (sec)	Accuracy	Time (sec)	Accuracy
500	179.65	55.27%	1,619.50	55.80%
1,000	342.34	55.28%	3,690.09	55.75%
1,500	507.45	55.27%	5,222.25	55.75%
2,000	664.25	55.27%	8,683.49	55.75%

(b) c14B

#{trees}	TreeServer		Spark MLlib	
	Time (sec)	Accuracy	Time (sec)	Accuracy
500	214.88	63.00%	981.01	49.83%
1,000	392.17	63.00%	1,914.00	49.82%
1,500	573.35	64.00%	2,849.23	49.83%
2,000	744.86	63.00%	4,126.12	49.81%

(c) XGBoost

#{trees}	MS_LTRC		c14B	
	Time (sec)	Accuracy	Time (sec)	Accuracy
10	220.91	57.28%	465.06	52.10%
20	413.62	57.46%	838.86	52.79%
40	738.99	57.99%	1,600.49	54.05%
80	1,330.58	58.42%	2,827.39	55.11%
100	1,585.10	58.83%	3,492.70	55.85%

20 trees, so our default setting $n_{pool} = 200$ is more than enough to fully utilize CPU resources. Also, Tables III(a)–(c) show that increasing n_{pool} only slightly increases the memory usage. This is because most memory is used to hold data columns, and memory used by tasks of each additional active tree allowed is small.

Effect of τ_{dfs} and τ_D. Recall that TreeServer has two parameters τ_{dfs} (resp. τ_D) to control the granularity for depth-first node-task scheduling (resp. granularity of subtree-tasks). If τ_{dfs} is too small (resp. large), there are not enough tasks for parallel execution initially (resp. compute-intensive subtree-tasks are not timely scheduled). Also, if τ_D is too small (resp. large), the subtree-tasks are too small to saturate the CPU cores (resp. too few to enable load balancing). We have tuned them and found the default setting ($\tau_D = 10,000$, $\tau_{dfs} = 80,000$) to work well on all our datasets. To illustrate, Tables III(d)–(e) show the performance on *Allstate*, *Higgs_boson* and *KDD99* where we fix one parameter as default and tune the other one.

Scalability to the Number of Trees. We now explore how the running time of a TreeServer job and that of an MLlib job change with the number of trees to train. For this purpose, we trained a random forest with 500, 1,000, 1,500 and 2,000 trees, respectively. We find that MLlib is too time-consuming to train on moderate-sized datasets such as *Allstate* where 7,778.74 seconds are spent to train just a 500-tree forest, for which TreeServer takes only 1736.78 seconds. To keep MLlib's time tractable for experiments, we focus on smaller datasets *MS_LTRC* and *c14B*, and the results are reported in Tables IV(a) and (b). As expected, the running time is proportional to the number of trees in both systems, since the large number of trees saturated the CPU core utilization; also, TreeServer is much faster than MLlib in all the experiments.

Note that using many trees does not impact a lot the accuracy in Tables IV(a) and (b). In contrast, boosting has more potential to improve the accuracy with more trees, as shown in Table IV(c) where XGBoost is trained with different number of trees, and the test accuracy keeps improving. However, since XGBoost is very expensive when the tree number gets large, we cannot test too many trees as in Tables IV(a) and (b).

TABLE V
VERTICAL SCALABILITY

(a) TreeServer (20 Trees)

#{threads}	Allstate	Higgs_boson
1	381.22 s	314.28 s
2	212.29 s	193.10 s
4	131.92 s	146.13 s
8	115.86 s	145.73 s
10	114.22 s	132.12 s

(b) Spark MLlib (20 Trees)

#{threads}	Allstate	Higgs_boson
1	1,635.96 s	985.92 s
2	945.45 s	565.90 s
4	558.73 s	350.39 s
8	401.73 s	266.17 s
10	305.86 s	259.75 s

(c) TreeServer (200 Trees)

#{threads}	Higgs_boson	MS_LTRC
1	2,510.49 s	271.10 s
2	1,340.00 s	141.50 s
4	777.62 s	80.02 s
8	738.16 s	73.27 s
10	755.73 s	74.51 s

(d) Spark MLlib (200 Trees)

#{threads}	Higgs_boson	MS_LTRC
1	11,096.97 s	1,027.33 s
2	5,977.08 s	575.48 s
4	3,515.03 s	572.77 s
8	2,120.65 s	571.77 s
10	1,978.36 s	556.71 s

Vertical Scalability. Recall that we run 10 compers on each machine by default. An interesting question to ask is whether 10 cores are fully utilized, and our finding is that the efficiency of TreeServer improves well as the number of compers increases, thanks to the subtree-tasks that are compute-heavy.

Tables V(a)–(d) shows the vertical scalability of TreeServer and MLlib when training a random forest with 20 and 200 trees. We illustrate using the first two datasets *Allstate* and *Higgs_boson*, but in the 200-tree scenario, running MLlib on *Allstate* is too time-consuming (3,102 seconds even with 10 threads) so we replace it with the third dataset *MS_LTRC* instead. We can see that the running time clearly reduces as we use more threads in both systems, but MLlib takes a few times more time than TreeServer in all cases.

Horizontal Scalability. Now that we know that each machine can scale with compers/threads, the next question to ask is how TreeServer scales out. This is an important question since if each machine is able to utilize all CPU cores well, then more computing power can be achieved by using more machines.

Table VI repeats our experiments in Tables II(a)–(d) but with different number of machines. we can see that a job of TreeServer is really compute-intensive and can keep multiple machines busy on task computation as we increase machine number from 4 to 12, and clear improvements can be observed even from 12 to 15 machines. For example, as Table II(a) shows, when building 1 tree on *Allstate* using TreeServer

TABLE VI
HORIZONTAL SCALABILITY (TIME UNIT: SECOND; NET UNIT: MBPS)

(a) TreeServer (1 Tree)

#{macs}	Allstate			Higgs_boson		
	Time	CPU	Send	Time	CPU	Send
4	123.76	837%	605.61	174.64	860%	940.40
8	76.72	760%	916.41	107.16	881%	942.32
12	65.74	699%	941.26	103.48	800%	942.52
15	61.93	694%	941.34	120.91	817%	942.32

(b) Spark MLlib (1 Tree)

Allstate	Higgs_boson
241.77	167.99
244.13	180.93
195.78	134.45
195.99	139.52

(c) TreeServer (20 Tree)

#{macs}	Allstate			Higgs_boson		
	Time	CPU	Send	Time	CPU	Send
4	236.79	726%	893.49	314.28	885%	941.33
8	146.26	710%	935.45	193.10	840%	941.48
12	142.78	716%	941.48	146.13	810%	941.28
15	118.18	663%	941.28	145.73	815%	942.25

(d) Spark MLlib (20 Tree)

Allstate	Higgs_boson
366.51	355.69
388.22	340.00
304.52	263.09
308.25	259.75

TABLE VII
DEEP FOREST EXPERIMENTAL RESULTS

Multi-Grained Scanning

Step	Training Time	Test Time
slide	163.58 s	34.95 s
win3train	716.57 s	–
win5train	971.93 s	–
win7train	1251.33 s	–
win3extract	105.00 s	38.69 s
win5extract	127.72 s	45.06 s
win7extract	117.85 s	42.57 s

Cascade Forest

Step	Training Time	Test Time	Test Accuracy
CF0train	94.88 s	–	–
CF0extract	41.04 s	27.07 s	96.87%
CF1train	99.31 s	–	–
CF1extract	41.95 s	28.08 s	96.82%
CF2train	93.55 s	–	–
CF2extract	39.63 s	27.44 s	97.68%
CF3train	94.73 s	–	–
CF3extract	37.68 s	26.84 s	98.10%
CF4train	91.61 s	–	–
CF4extract	41.99 s	27.93 s	97.02%
CF5train	92.46 s	–	–
CF5extract	40.07 s	27.63 s	97.71%

running 10 compers, we can achieve 837% average CPU rate (meaning that 8 cores are fully utilized). As we increase machine number to 15, CPU rate is still 694% but since more data transmission is required, average sending throughput saturates at 941.34 Mbps (we use 1 GigE) so performance improvement flattens. We expect the performance to improve further with more machines if 10 GigE is available. As Tables II(a) and (c) illustrates, we are able to consistently keep > 6 (oftentimes > 8) CPU cores busy while fully utilizing the available network bandwidth. In contrast, MLlib sees less significant improvement and is more expensive in all cases.

Experiments on Deep Forests. We run our deep forest pipeline built with TreeServer on the MNIST dataset which consists of 28×28 gray images, 60,000 (resp. 10,000) of them are for training (resp. testing). All images are for digits 0–9, ten classes in total. We used MGS window size 3×3, 5×5 and 7×7, and we followed Table 1 of [34] to set the hyperparameters but make the following changes to improve accuracy: (1) extra trees reduce the test accuracy when used in CF so we only use random forests for CF, (2) setting $d_{max} = 100$ in MGS drops the test accuracy so we use 10, (3) to reach good accuracy, each step only needs 2 forests each with 20 trees (rather than 500 trees), and only 10% of the training and test images are needed and thus used.

Table VII reports the running time of each step as illustrated in Fig. 11, as well as their corresponding test accuracy. Specifically, "slide" corresponds to the step that slides each image to generate inputs to MGS. In MGS, "win3train", for example, corresponds to the step that trains the forests over vectors extracted by 3×3 windows, and "win3extract"

TABLE VIII
IMPACT OF MODEL PARAMETERS

(a) Effect of d_{max} on Higgs_boson (1 tree)

d_{max}	Time	Accuracy
2	69.47 s	63.117%
4	93.37 s	65.635%
6	118.62 s	67.946%
8	112.96 s	69.456%
10	115.86 s	70.515%
12	113.04 s	71.313%

(b) Effect of d_{max} on Higgs_boson (20 trees)

d_{max}	Time	Accuracy
2	84.65 s	62.372%
4	115.99 s	61.571%
6	124.21 s	62.655%
8	128.73 s	63.546%
10	155.68 s	63.893%
12	170.11 s	64.153%

(c) Effect of $|C|/|A|$ on Allstate

$\|C\|/\|A\|$	Time	RMSE
20%	121.79 s	41.460
40%	145.71 s	41.469
60%	220.35 s	41.462
80%	241.02 s	41.463
100%	255.14 s	41.464

(d) Effect of $|C|/|A|$ on Higgs_boson

$\|C\|/\|A\|$	Time	Accuracy
20%	158.66 s	63.893%
40%	266.23 s	70.355%
60%	416.07 s	71.038%
80%	423.70 s	71.506%
100%	622.49 s	70.515%

corresponds to the step that used the trained forests to represent each input image. In the CF stage, "CF0train", for example, corresponds to the time to train the forests of CF0 while "CF0extract" corresponds to the time to use the trained forest in CF0 to represent the input images. Note that after each CF step, we report our test accuracy which are all very high and improves all the way up to CF3. We observe that training accuracy is always 100% because the maximum tree depth in the CF stage is $d_{max} = \infty$, so we omit them in Table VII. We can see that training is very efficient despite the many trees.

Fairness of Implementation. Since MLlib runs on JVM, while our TreeServer program runs as a native C++ program, it is worth exploring whether our improvement comes from the language rather than the system design. For this purpose, we run single-threaded single-tree construction for both TreeServer and MLlib and find that TreeServer is comparable to MLlib. For example, TreeServer takes 705.94 seconds while MLlib takes 750.58 seconds on *Higgs_boson*, and TreeServer takes 191.86 seconds while MLlib takes 157.34 seconds on *MS_LTRC*. Our single-tree implementation is not faster since in order to allow flexible user data input like in pandas, we conduct runtime (rather than compile-time) type conversion based on the data type of each column being processed.

Accuracy w.r.t. Model Parameters. Since TreeServer implements the exact node splitting algorithms, we obtain exact models as in a conventional serial algorithm but runs much faster. Table VIII(a) (resp. (b)) shows the time and test accuracy of training a tree (resp. forest with 20 trees) on *Higgs_boson* with increasing d_{max} and we can see that the accuracy keeps improving (i.e., models are not overfitting). Tables VIII(c)–(d) show the performance of training a 20-tree forest with different $|\mathcal{C}|$ sampled for each tree on *Allstate* and *Higgs_boson*, where the test accuracy does not change a lot meaning that 20% columns per tree are already sufficient.

IX. CONCLUSION

We presented a distributed system called TreeServer for training tree models, which addresses the problems of IO bottleneck in existing solutions such as MLlib. The system adopts novel designs including column-based data partitioning and node-centric task-based workload partitioning, and is consistently many times faster than MLlib, hence a good alternative especially for training big models like a deep forest.

REFERENCES

[1] Allstate Dataset. https://www.kaggle.com/c/allstate-claims-severity/data.

[2] Forest CoverType Dataset. https://archive.ics.uci.edu/ml/datasets/covertype.

[3] Higgs Boson Dataset. https://archive.ics.uci.edu/ml/datasets/HIGGS.

[4] KDD Cup 1999 Dataset. https://archive.ics.uci.edu/ml/datasets/kdd+cup+1999+data.

[5] Microsoft Learn2Rank Dataset. https://www.microsoft.com/en-us/research/project/mslr/.

[6] Poker Hand Dataset. https://archive.ics.uci.edu/ml/datasets/poker+hand.

[7] Single Family Loan-Level Dataset. http://www.freddiemac.com/research/datasets/sf_loanlevel_dataset.page.

[8] Supersymmetry (SUSY) Dataset. https://archive.ics.uci.edu/ml/datasets/SUSY.

[9] TreeServer Demo (Long Version). https://www.youtube.com/watch?v=4DnLv_OFlIg.

[10] TreeServer Demo (Short Version). https://www.youtube.com/watch?v=IawT40DhGbs.

[11] TreeServer GitHub Repository. https://github.com/yanlab19870714/TreeServer.

[12] TreeServer's Deep Forest Demo. https://www.youtube.com/watch?v=LzXkzkk0r_0.

[13] TreeServer's Deep Forest GitHub Repo. https://github.com/yanlab19870714/deepForest.

[14] Yahoo Learn2Rank Dataset. https://webscope.sandbox.yahoo.com/catalog.php?datatype=c.

[15] F. Abuzaid, J. K. Bradley, F. T. Liang, A. Feng, L. Yang, M. Zaharia, and A. S. Talwalkar. Yggdrasil: An optimized system for training deep decision trees at scale. In *NIPS*, pages 3810–3818, 2016.

[16] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*. Wadsworth, 1984.

[17] T. Chen and C. Guestrin. Xgboost: A scalable tree boosting system. In *SIGKDD*, pages 785–794, 2016.

[18] W. Fan, H. Wang, P. S. Yu, and S. Ma. Is random model better? on its accuracy and efficiency. In *ICDM*, pages 51–58, 2003.

[19] J. Feng, Y. Yu, and Z. Zhou. Multi-layered gradient boosting decision trees. In *NeurIPS*, pages 3555–3565, 2018.

[20] Frank McSherry. COST in the land of databases. https://github.com/frankmcsherry/blog/blob/master/posts/2017-09-23.md, 2017.

[21] G. Guo, D. Yan, M. T. Özsu, and Z. Jiang. Scalable mining of maximal quasi-cliques: An algorithm-system codesign approach. *PVLDB*, 2021.

[22] J. Khalil, D. Yan, G. Guo, and L. Yuan. Parallel mining of large maximal quasi-cliques. *The VLDB Journal (to appear)*, 2021.

[23] Q. Meng, G. Ke, T. Wang, W. Chen, Q. Ye, Z. Ma, and T. Liu. A communication-efficient parallel algorithm for decision tree. In *NIPS*, pages 1271–1279, 2016.

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

[25] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing, Philadelphia, Pennsylvania, USA, May 23-26, 1996*, pages 267–275, 1996.

[26] A. Musselman. Apache mahout. In *Encyclopedia of Big Data Technologies.* 2019.

[27] B. Panda, J. Herbach, S. Basu, and R. J. Bayardo. PLANET: massively parallel learning of tree ensembles with mapreduce. *PVLDB*, 2(2):1426–1437, 2009.

[28] W. Qu, D. Yan, G. Guo, X. Wang, L. Zou, and Y. Zhou. Parallel mining of frequent subtree patterns. In *LSGDA@VLDB*, 2020.

[29] D. Yan, G. Guo, M. M. R. Chowdhury, M. T. Özsu, W.-S. Ku, and J. C. S. Lui. G-thinker: A distributed framework for mining subgraphs in a big graph. In *ICDE*, 2020.

[30] D. Yan, G. Guo, M. M. R. Chowdhury, M. T. Özsu, J. C. S. Lui, and W. Tan. T-thinker: a task-centric distributed framework for compute-intensive divide-and-conquer algorithms. In *PPoPP*, pages 411–412, 2019.

[31] D. Yan, G. Guo, J. Khalil, M. T. Özsu, W.-S. Ku, and J. Lui. G-thinker: a general distributed framework for finding qualified subgraphs in a big graph with load balancing. *The VLDB Journal*, pages 1–34, 2021.

[32] D. Yan, W. Qu, G. Guo, and X. Wang. Prefixfpm: A parallel framework for general-purpose frequent pattern mining. In *ICDE*, 2020.

[33] D. Yan, W. Qu, G. Guo, X. Wang, and Y. Zhou. Prefixfpm: a parallel framework for general-purpose mining of frequent and closed patterns. *The VLDB Journal*, pages 1–34, 2021.

[34] Z. Zhou and J. Feng. Deep forest: Towards an alternative to deep neural networks. In *IJCAI*, pages 3553–3559, 2017.

Appendix

A. List of Notations

The paper used the following notations, which are summarized for quick reference as needed:

- D: the input data table;
- n: number of rows in the data table;
- m: number of attributes (i.e., columns) in the data table;
- A_i: the i-th attribute in the data table;
- $\mathcal{A}$: the set of all attributes $A_1, A_2, \ldots, A_m$;
- Y: the target attribute to predict;
- x: a node in a decision tree;
- Δ_x: the subtree rooted at node x;
- D_x: those rows of the training data that reach node x following the decision path from the root node;
- $|D_x|$: the number of rows in D_x;
- I_x: the row-IDs of those rows in D_x;
- v: the split-value of a numerical attribute A_i, used in split-condition "$A_i \leq v$";
- S_i: the set of all possible values of a categorical attribute A_i;
- S_ℓ: a subset of attribute values in S_i which lead a data to go to the left child node of x, used in split-condition "$A_i \in S_\ell$";
- $\mathcal{C}$: a subset of $\mathcal{A}$, which restricts the split-condition of node x to be chosen only from an attribute in $\mathcal{C}$;
- $|\mathcal{C}|$: a number of columns in $\mathcal{C}$;
- t_x: the task that processes node x to construct its subtree;
- x_ℓ: the left child node of node x;
- x_r: the right child node of node x;
- τ_D: a system threshold such that if $|D_x| \leq \tau_D$, the entire subtree Δ_x is built by a subtree-task t_x;
- τ_{dfs}: a system threshold such that if $|D_x| \leq \tau_{dfs}$, nodes in the subtree Δ_x are processed in depth-first order (breadth-first order otherwise), and usually $\tau_{dfs} > \tau_D$ (see Fig. 4);
- n_{pool}: the maximum number of trees allowed to be under construction at any time;
- B_{plan}: a buffer (deque) in the master to hold tasks (aka. plans) newly created but not assigned worker workloads yet;
- d_{max}: the maximum allowed tree depth, which is a model hyperparameter;
- τ_{leaf}: a model hyperparameter such that a node x stops its splitting to avoid overfitting when $|D_x| \leq \tau_{leaf}$;
- Q_{plan}: a plan queue in the master to buffer task-plans that have been assigned worker workloads, and these plans will be sent to the destination workers in batches;
- T_{task}: a task table to keep tasks that are being processed but are waiting for dependent data;
- θ_{main}: the main thread in a machine;
- θ_{recv}: a communication thread that receives responses from other machines;
- comper: a computing thread in a worker machine;
- B_{task}: a task buffer of a worker machine from which compers take tasks for computation;
- $\mathcal{M}_{work}$: a workload matrix that the master uses to track worker workloads, where each row i is for a worker, and $\mathcal{M}_{work}[i][Comp]$, $\mathcal{M}_{work}[i][Send]$ and $\mathcal{M}_{work}[i][Recv]$ tracks Worker i's computation, sending and receiving workloads, respectively.

B. Selecting Best Split-Condition for a Table Column

In decision tree training, at each tree node x, we select the best split-condition for each individual attribute (column) A_i independently. Depending on the type of A_i and Y, there are three cases:

- **Case 1: A_i is an ordinal attribute.** To find the best value v for the split-condition "$A_i \leq v$", only the values of A_i that appear in D_x are considered. It is well known that by sorting rows of D_x by A_i-values and checking each value v to split the rows into left and right child nodes, we can get the impurity value for each v in $O(1)$ incremental cost and thus we can find the best v in one pass over the sorted D_x.
- **Case 2: A_i is categorical, and Y is numerical.** This case is for regression, and Breiman et al. [16] present an algorithm for finding the best split predicate without evaluating all possible subsets of S_i for S_ℓ. After grouping rows of D_x by A_i-values and sorting the groups by the average Y-value, the optimal split predicate cuts the sorted group list in the middle, so the algorithm only needs one pass over the groups.
- **Case 3: both A_i and Y are categorical.** This case is for classification, and we have to enumerate and check all possible subsets of S_i for S_ℓ to split rows in D_x. To limit the checking cost when $|S_i|$ is big, it is common to restrict $|S_\ell| = 1$ so that only $O(|S_i|)$ split-conditions need to be checked.

C. Tree Construction Progress Tracking

The master uses a progress table $\mathcal{T}_{prog}$ to track the number of pending tasks for each active tree under construction, so that once it receives the results from the tree's last task and completes the tree construction, it can reduce the count of active trees so that new trees can be admitted for processing (recall that TreeServer executes a budget of at most n_{pool} active trees at any moment).

In our implementation, each task keeps its tree ID tid. Here, a tree "root" task is created by setting its tid properly when the tree is admitted for processing, while each descendent node-centric task t_x inherits its tree ID tid from its parent task $t_{pa(x)}$, where $pa(x)$ is the parent node of node x.

Each task sends its tree ID back to the master along with the task result, to update the progress counter $\mathcal{T}_{prog}[tid]$: incremented if the task is split into two child tasks, and decremented otherwise. This is because a column-task is consumed by the master which creates two new child tasks, leading to a net increment of 1; while a subtree-task is consumed without creating new tasks, leading to a net increment of -1.

As soon as the master finds that $\mathcal{T}_{prog}[tid]$ is decremented to 0 after processing the result of a task from Tree tid, it knows

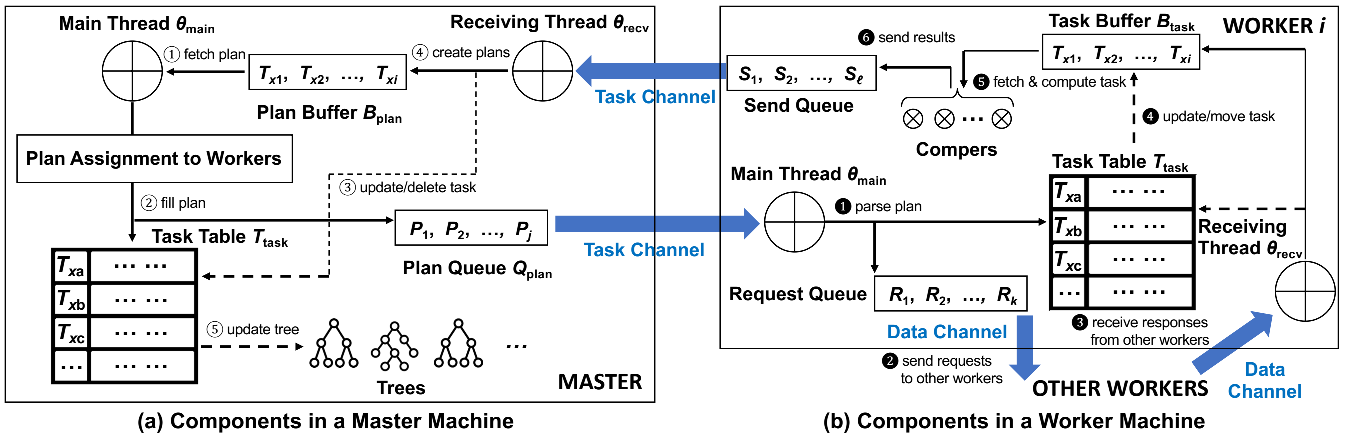


Fig. 14. Components of Master and Other Machines

that the last task of Tree tid is completed so it can output the tree and update the bookkeeping information accordingly to reflect the completion of constructing Tree tid.

D. Other TreeServer Features

TreeServer automatically infers the most appropriate algorithm (run by a column-task) to find the best split-condition of each attribute A_i based on its data type, so the input data is totally flexible. Also, while a decision tree usually maintains the predicted labels or PMF vectors only at leaf nodes, we also let internal nodes maintain them. Note that the predicted labels or PMF vectors can be easily computed as a byproduct since each node x has access to D_x during training.

When predicting the label of a new entity, this design allows the search to stop at any depth to output the predicted label at the current node. Therefore, if we train a tree with maximum depth d_{max}, we can use it for prediction as a tree with maximum depth anywhere from 1 to d_{max} without the need to train those trees.

This design also handles new attribute values unseen during training. Specifically, for a test entity, when we visit a node x whose split-condition involves attribute A_i, and find that the entity's A_i-value is never unseen in D_x during training, we then treat node x as a leaf directly to report its predicted label or PMF vector for the current entity. This is because it is unreasonable for the entity to go to either child of x.

The same applies when we encounter a missing value for attribute A_i when we visit a node x for a test entity, in which case we directly report x's predicted label or PMF vector for the current entity.

E. T-thinker Components & Thread Workflow

Hereafter, we call a task t_x not yet sent to workers for processing as a **plan**. In master, all newly created plans that have not been assigned for processing are kept in the plan buffer B_{plan} (see Fig. 14(a)) implemented as a deque, while after the workers that will process a plan t_x are determined, t_x is placed into a plan queue Q_{plan} (see Fig. 14(a)) to be sent to workers for processing. Note that a new plan in B_{plan} may generate multiple column-task plans in Q_{plan} directed to different workers that collectively hold columns $A_i \in \mathcal{C}$.

Fig. 14 shows the components for task processing in the master (left) and worker machines (right). Besides deque B_{plan} protected with mutex, a task table T_{task} is implemented as a concurrent hash table so that the insertion and fetching of different tasks may proceed concurrently as long as they are not in the same bucket, while Q_{plan}, B_{task} and all message queues are implemented as concurrent queues [25] that support simultaneous enqueue and dequeue operations.

We next explain the thread workflows in the master and workers, respectively.

Thread Workflows in Master. Master has two key threads: (1) main thread θ_{main} and (2) receiving thread θ_{recv}.

Thread θ_{main} loops the following operations. It fetches a plan from B_{plan} to compute its worker assignment, i.e., operation ① in Fig. 14(a). Meanwhile, if the tasks in B_{plan} belong to less than n_{pool} trees, θ_{main} replenishes the next unprocessed "root"-node task into to B_{plan} to start its processing.

If θ_{main} obtains a plan t_x from B_{plan}, it computes the worker assignment using the algorithm described in Section VI, and then appends the resulting plan(s) to a plan queue Q_{plan} (see Fig. 14(a)) that sends plans to workers in batches. While a plan t_x is being processed by workers, the task information of t_x is also inserted into the master's task table T_{task} waiting for the computation results to be sent back. These correspond to operation ② in Fig. 14(a).

If θ_{main} cannot find a plan in B_{plan}, it sleeps for 100 μs to avoid busy waiting before probing B_{plan} again. The loop terminates when all trees have been processed and B_{plan} is empty, after which θ_{main} flags all other threads at the master side to terminate their queue probing loop; it also inserts special messages in Q_{plan} to notify workers to terminate.

The other key thread θ_{recv} processes each received message that contains the computed result of a task t_x, and uses it to update the table entry of t_x in T_{task}; if t_x is determined to be finished, θ_{recv} further updates the tree under construction using the task result, removes t_x from T_{task} and deletes the task object t_x (see operations ③ and ⑤ in Fig. 14(a)).

A completed column-task t_x will generate two child task-plans t_{x_ℓ} and t_{x_r} (unless x is a leaf), and they are added to the plan buffer B_{plan} to be scheduled for processing. This process is illustrated by operations ④ in Fig. 14(a). To prevent premature exit of θ_{main}, we require θ_{recv} to always add plans t_{x_ℓ} and t_{x_r} into B_{plan} before decrement the tree-counter of t_x to indicate the completion of t_x, so that B_{plan} is never empty before t_x's tree is fully constructed.

Thread Workflows in a Worker. As Fig. 14(b) shows, a worker runs a main thread θ_{main} to receive task-plans from the master, a receiving thread θ_{recv} that prepares data for tasks, and a pool of computing threads (**compers**) to compute tasks.

A task-plan is processed in 5 steps. ❶: θ_{main} keeps receiving plan-messages from the master, until a termination notification is received to terminate the worker program. For each plan-message, θ_{main} parses it to decide if it is a subtree-task or a column-task. A subtree-task t_x still needs to obtain its data D_x from other workers, so θ_{main} poses those data requests to a request queue for sending, as shown by operation ❷ in Fig. 14(b); θ_{main} also puts t_x in the task table T_{task} waiting for the data responses. ❸: when θ_{recv} receives the data requested by t_x, it attaches the data to t_x's entry in T_{task}; if all necessary data are now with t_x, θ_{recv} will further move t_x from T_{task} to B_{task} to be fetched by a comper for processing. This process is illustrated by operation ❹ in Fig. 14(b). ❺: compers concurrently fetch tasks from B_{task} for computation; each comper keeps fetching and computing tasks until θ_{main} flags termination, and if there is no task in B_{task}, the comper will sleep for 100 μs to avoid busy waiting. ❻: the computed results are appended by the compers to a sending queue which delivers them to the master in batches.

Fault Tolerance. Since a TreeServer program is master-driven, the master is the only single point of failure which can be strengthened by enabling a secondary master. Specifically, a worker can act as a secondary master that periodically communicates with the master to check if it is reachable. If not, the secondary master will send a special message to the other workers notifying them that it is now the new master so that they will direct task-channel messages to the secondary master. When the secondary master is enabled, the master needs to periodically synchronize the job metadata and tree construction progress to the secondary master. New tasks assigned since the last synchronization will be reassigned by the secondary master, which accepts but ignores old responses.

If a worker crashes, the master simply reassigns its lost columns to other machines by copying from the column replicas, and for each task in task table T_{task} whose computation involves the crashed worker, the master notifies workers to revoke these tasks and to delete their task objects; we also move these tasks in the master from T_{task} back to the head of B_{plan} so that their new workers can be reassigned ASAP.

F. Extra-Tree Support in TreeServer Deep Forests

Besides random forests, deep forest also uses "completely random decision trees" (aka. extra-trees) as the other forest type: such a tree resamples a column A_i from all attributes after each node splitting, and randomly samples a splitting value v from $[min, max]$ where min and max are the smallest and largest A_i-values in D_x, respectively. TreeServer also supports extra-trees, where the only difference from building random forests is that a subtree-task needs to get data from all columns (of rows I_x) for subsequent column-sampling after each node-splitting when building the subtree, rather than simply obtain those columns of $\mathcal{C}$ as in a random forest.

G. The Loan Dataset

The dataset *loan* [7] has two tables: "Origination Data" describing information of each loan, and "Monthly Performance Data" describing the monthly loan payment information. The two tables were joined on the attribute "LOAN SEQUENCE NUMBER" to obtain the final data table D. Since D has a lot of missing data, we removed every column with more than 75% missing values, and cleansed the rest by filling missing values with the mean attribute value. We also took the loan data from the latest month, the latest year, and the latest 2 years to obtain 3 datasets of different sizes as shown in Table I.