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. [17] 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.

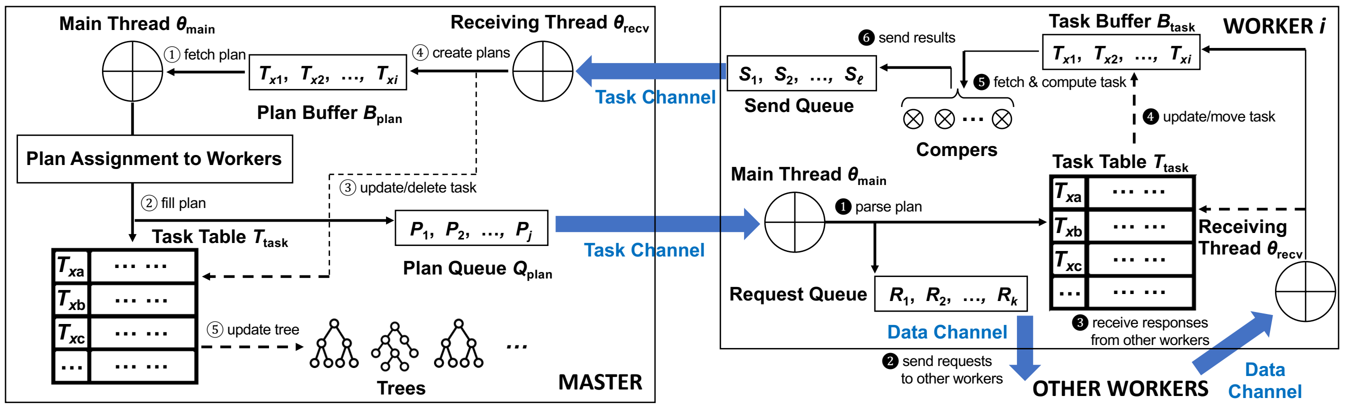
C. Tree Construction Progress Tracking

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



(a) Components in a Master Machine (b) Components in a Worker Machine

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 [28] 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* [8] 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.