Eliminating Redundant Feature Tests in Decision Tree and Random Forest Inference on SQL Predicates

MINGXI LIU, East China Normal University[†‡], China
ZHENGYUAN DING, East China Normal University[†‡], China
CHENYANG ZHANG, East China Normal University[†‡], China
QINGFENG PAN, East China Normal University[†‡], China
HUAYOU SU, National University of Defense Technology, China
ZHAO ZHANG, East China Normal University[†‡], China
CHEN XU[*], East China Normal University[†‡], China
QINGSONG RUAN, China Electronics Technology Kingbase (Beijing) Technologies Inc., China

In-database prediction queries that apply machine learning (ML) pipelines to perform data analysis are prevalent in many applications. Since data stored in databases is typically tabular, tree-based models are particularly well-suited and thus widely adopted for such tasks. When ML inference with a decision tree or random forest appears on a SQL predicate, existing works first perform ML inference and then determine whether the inference result satisfies the predicate. However, this leads to redundant feature tests on the tree node during the predicate evaluation. To determine whether one data record satisfies the predicate, it is possible to perform feature tests only on partial internal nodes of the tree. We identify that these redundant feature tests are caused by specific sibling and ancestor nodes. In particular, we propose the sibling-centric elimination with the *merging-based subtree collapse* method, and the ancestor-centric elimination with the *sliding-based subtree recombination* method. We implement a prototype system, called ReTree, based on DuckDB. Our experiments show that ReTree achieves a 2.56x speedup on average over DuckDB for prediction query execution and outperforms other solutions.

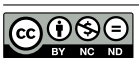
CCS Concepts: • **Information systems** → **Query planning**; **Query optimization**; • **Computing methodologies** → **Machine learning**.

Additional Key Words and Phrases: Query Planning, Query Optimization, Tree-based Model Inference

ACM Reference Format:
Mingxi Liu, Zhengyuan Ding, Chenyang Zhang, Qingfeng Pan, Huayou Su, Zhao Zhang, Chen Xu, and Qingsong Ruan. 2026. Eliminating Redundant Feature Tests in Decision Tree and Random Forest Inference on SQL Predicates. *Proc. ACM Manag. Data* 4, 3 (SIGMOD), Article 171 (June 2026), 27 pages. https://doi.org/10.1145/3802048

[†]Engineering Research Center of Blockchain Data Management (East China Normal University), Ministry of Education
[‡]Shanghai Engineering Research Center of Big Data Management
[*]Chen Xu is the corresponding author

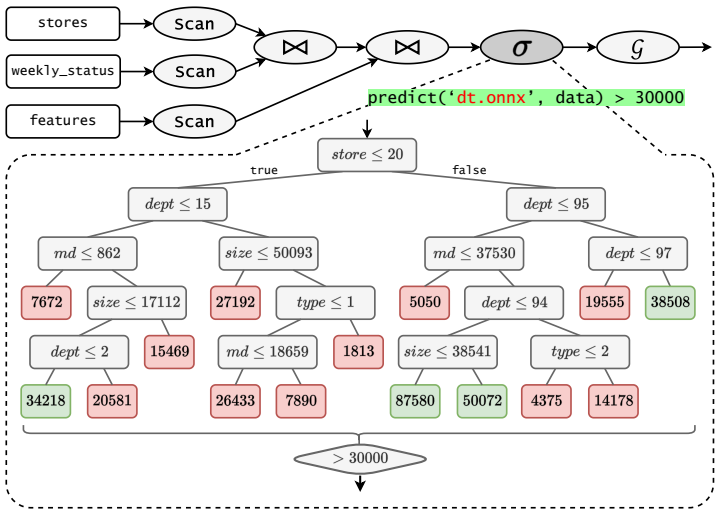
Authors' Contact Information: Mingxi Liu, East China Normal University, Shanghai, China, mxliu@stu.ecnu.edu.cn; Zhengyuan Ding, East China Normal University, Shanghai, China, zyding@stu.ecnu.edu.cn; Chenyang Zhang, East China Normal University, Shanghai, China, cyzhangecnu@stu.ecnu.edu.cn; Qingfeng Pan, East China Normal University, Shanghai, China, qfpan@stu.ecnu.edu.cn; Huayou Su, National University of Defense Technology, Changsha, China, shyou@nudt.edu.cn; Zhao Zhang, East China Normal University, Shanghai, China, zhzhang@dase.ecnu.edu.cn; Chen Xu, East China Normal University, Shanghai, China, cxu@dase.ecnu.edu.cn; Qingsong Ruan, China Electronics Technology Kingbase (Beijing) Technologies Inc., Beijing, China, qsruan@kingbase.com.cn.



ACM 2836-6573/2026/6-ART171
https://doi.org/10.1145/3802048

```
SELECT count(*) AS weeks
FROM weekly_status w JOIN stores s ON w.store = s.store
JOIN features f ON w.store = f.store AND w.date = f.date
WHERE predict('dt.onnx', data) AS weekly_sales > 30000
GROUP BY w.store, w.dept;
```

(a) SQL Query Statement



(b) Unified IR based Query Plan

Fig. 1. Example Query with Decision Tree Predicate

1 Introduction

Nowadays, machine learning (ML) is widely used in various fields for data analysis [42], such as fraud transaction detection [5]. Prediction queries [15] are analytical queries that perform ML inference on data using trained ML models. Typically, for enterprise applications, key tabular data is stored in relational databases [23, 31, 55, 56]. However, common practice that exploits user scripts to perform prediction queries requires exporting data from databases first, which incurs additional overhead of data transfer and materialization [11, 15, 19, 29, 30, 58]. In addition, for privacy and security constraints, data is not allowed to be moved out of databases [18]. Hence, it is important to perform prediction queries directly inside the database, thereby supporting real-time data analysis.

When ML inference appears on a SQL predicate, we call it an *ML predicate* [10, 15]. A typical ML predicate involves applying a trained model to specific columns, followed by a comparison operator and a constant. Tree-based models, such as decision trees and random forests, are extensively used for tabular data [1, 8, 9, 13, 35, 45, 50] and are often employed to filter relevant records from raw data tables [10, 15, 31, 41, 58]. In this work, we focus on optimizing two common types of ML predicates, i.e., *decision tree predicates* and *random forest predicates*, which employ decision tree and random forest models, respectively.

Example 1. As the prediction query shown in Figure 1(a), the green part is an ML predicate. The query is executed on weekly data from Walmart stores [14], which are stored in three tables: `stores`, `features`, and `weekly_status`. The query aims to find, for each store and department, how many weeks have predicted sales exceeding $30,000. It first collects features by joining the

three tables, then predicts the weekly sales for each department in each store using an ML pipeline with a decision tree from the model file `dt.onnx`. Then, data records with inference results greater than 30,000 are selected for aggregation.

Figure 1(b) shows the query plan of Example 1, which includes a decision tree predicate. The tree uses five features from the joined table: the store number *store*, the department number *dept*, promotional markdown events *md*, and the *size* and *type* of stores. During execution, each data record is evaluated through a sequence of *feature tests* [36]. At each internal node of the tree, the test outcome determines which child to follow from a pair of *sibling nodes* until the record reaches a leaf node. The leaf and its ancestor nodes together form the *decision path*, and the leaf value is the inference result, which is then compared with 30,000. However, Example 1 does not require the exact inference result but only needs to determine whether the inference result satisfies the predicate. This observation leads to a key insight: *for one data record, it is possible to determine whether the inference result satisfies the predicate early, without traversing the full decision path to a leaf node.* Hence, feature tests on internal tree nodes may be redundant, which increases the overhead of evaluating ML predicates.

To reduce the overhead of ML predicate evaluation, existing work has proposed various approaches. PP [21] and CORE [54] add predicates using lightweight proxy models before expensive ML predicates. Similarly, Smart [10] adds simple comparison expression predicates derived from the parameters of linear or tree-based models in ML predicates. These works reduce the amount of input data of the ML predicates without modifying the predicates. Differently, Raven [15, 31] simplifies ML predicates by pruning tree-based models using other predicates that share the same input columns. Consequently, for decision tree and random forest predicates, existing works are oblivious to the redundant feature tests from the perspective of the inference results, which often arise from certain sibling or ancestor nodes in the tree.

In this work, we present ReTree to eliminate redundant feature tests during the evaluation of decision tree and random forest predicates, and integrate it into an analytical database system DuckDB [38]. In particular, we find that these redundant feature tests are caused by specific sibling and ancestor nodes. To eliminate the sibling-induced redundant feature tests, ReTree uses sibling-centric elimination with the *merging-based subtree collapse* approach. With sibling-centric elimination, ReTree achieves a 2.14x performance improvement on average in comparison to DuckDB. Moreover, to eliminate the ancestor-induced redundant feature tests, ReTree uses ancestor-centric elimination with the *sliding-based subtree recombination* approach. As a result, it achieves an additional 1.15x speedup on average. Our extensive experiments in this paper show that ReTree accelerates the prediction query execution 2.56x on average over DuckDB, and outperforms alternative solutions like Raven [15, 31] and Smart [10].

In the remainder of this paper, we highlight the motivation of ReTree in Section 2. Then, we make the following contributions.

- We propose *sibling-centric elimination* in Section 3, which uses the *merging-based subtree collapse* method to eliminate the sibling-induced redundant feature tests.
- We propose *ancestor-centric elimination* in Section 4, which uses the *sliding-based subtree recombination* method to eliminate the ancestor-induced redundant feature tests.
- We discuss the extension to random forest predicates in Section 5 and the implementation of ReTree on DuckDB in Section 6.
- Our experiments demonstrate that ReTree significantly outperforms the state-of-the-art solutions in Section 7.

In addition, we discuss limitations and future work in Section 8, introduce related works in Section 9 and conclude in Section 10.

2 Background and Motivation

We introduce the background in Section 2.1, and illustrate our motivation in Section 2.2.

2.1 Background

ML Predicate. Modern prediction queries typically apply ML predicates in the SQL statement to filter out user-interested records from the original data table [2, 15, 25, 30, 31, 41, 58]. An *ML predicate* is an SQL predicate that applies a trained ML pipeline to input features and performs selection based on the inference result of the ML pipeline. In practice, the syntax form of the ML predicate widely used in SQL follows the pattern:

```
PREDICTION COMPARE CONST
```

Here, `PREDICTION` represents an inference function (i.e., the trained ML pipeline) , `COMPARE` is a comparison operator, and `CONST` refers to a comparison threshold against which the inference result is evaluated. For the query in Example 1 , the `PREDICTION` component is `predict('dt.onnx', data)`, the `COMPARE` operator is `GREATER`, and the `CONST` value is 30,000. Basically, in current databases, e.g., DuckDB and PostgreSQL, an SQL statement with an ML predicate is first translated into a query plan, where the ML predicate is parsed as an expression attached to the selection operator. During execution, the `PREDICTION` part is first computed and then compared with the `CONST` part, returning true or false. This comparison result is further used by the selection operator to filter qualifying data records.

Decision Tree Predicate. Tree-based models, e.g., decision trees and random forests, are widely adopted for regression and classification tasks in practice, especially on tabular data [1, 8, 9, 13, 35, 45, 50]. As a kind of ML predicate, *decision tree predicate* (DTP) is an ML predicate using an a decision tree model in the `PREDICTION` part. For one DTP, as depicted in Figure 1(b), there are two different kinds of nodes in the model inference part. One is internal nodes, i.e., the gray nodes in the figure, which perform *feature tests* of the form $feature \leq threshold$. The other is leaf nodes, i.e., the red and green nodes, which store the inference result. During inference, each data record starts at the root node and iteratively tests feature values against the thresholds of internal nodes. Following the convention of existing ML frameworks [26, 44], if the record satisfies the condition at a node, the process moves to the left child. Otherwise, it moves to the right child. This process continues until a leaf node is reached, whose value is returned as the inference result and subsequently compared with the `CONST` value to produce the comparison result of the DTP.

Evaluation Overhead. Since the model inference part accounts for the majority of DTP evaluation, we treat the overhead of tree inference as the DTP evaluation overhead. Given a data record, we define its *decision path* as the sequence of nodes traversed from the root to a leaf node, and its *length* as the number of internal nodes visited during inference. Since data records traverse paths of varying lengths depending on their feature values, we calculate the *averaged decision path length* over all records in a given table to measure the overall evaluation overhead of the decision tree predicate. Hence, we have

$$C = \sum_{i=1}^{n} f_i c_i \tag{1}$$

where C is evaluation overhead of the DTP, n is the number of leaf nodes, f_i is the frequency of data records reaching the i-th leaf node, and c_i is the number of feature tests required for a data record to reach the i-th leaf node from the root node.

Cross-Optimization with Unified IR. To enable cross-optimiza-tion, existing works typically transformed the query plan into a unified intermediate representation (IR), which captures the semantics of both relational algebra (RA) and ML operators. Specifically for the DTP, the IR consists

Table 1. Data Records for the Query in Example 1

ID	Columns (or Features)					# of Feature Tests
	store	*dept*	*md*	*size*	*type*	
❶	1	67	23,469	60,605	1	5
❷	21	95	49,370	36,530	2	5
❸	5	12	32,397	8,560	3	5
❹	10	15	1,398	51,215	1	4
❺	26	96	55,805	8,560	3	3

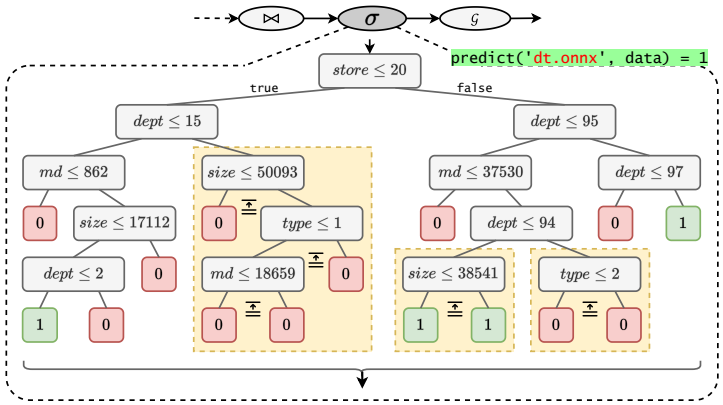
of two parts, i.e., the RA part and the decision tree model part, shown in Figure 1(b). By utilizing the unified semantics from the IR, Raven [15, 31] simplifies the decision tree model part, considering other relational predicates of RA part. Smart [10] generates additional simple relational predicates before ML predicates to reduce the number of records passed to model inference, leveraging the relationship between ML operators and database statistics. However, these works ignore the characteristic that evaluating the DTP only needs to acquire the final result of true or false and does not need to obtain the exact inference result of the model. This results in redundant feature tests from internal nodes during the evaluation. Hence, in this work, we focus on eliminating those redundant tests in DTP evaluation based on the unified IR.

2.2 Motivation

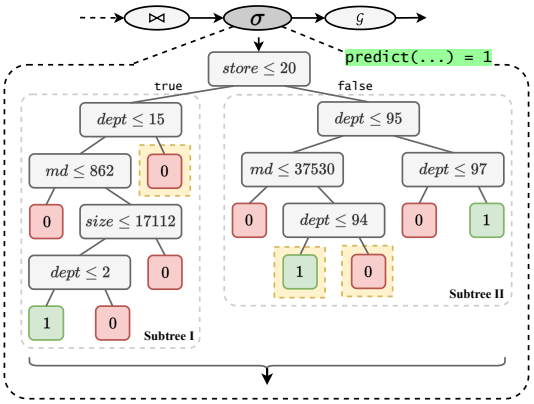
We highlight two types of redundant feature tests during decision tree predicate evaluation. Specifically, we introduce the sibling-induced redundant feature test and the ancestor-induced redundant feature test. For the query in Example 1, the DTP filters the data from the joined table of `features`, `weekly_status`, and `stores`, retaining only the data records of which predicted weekly sales are greater than $30,000 for further processing. Table 1 shows five data records from the joined table. During DTP evaluation of this query, each record follows a distinct decision path in the tree to produce its inference result.

Sibling-Induced Redundant Feature Test. Considering the decision path of Record ❶, i.e, $(store \leq 20) \rightarrow (dept \leq 15) \rightarrow (size \leq 50,093) \rightarrow (type \leq 1) \rightarrow (md \leq 18,659) \rightarrow 7,890$, it needs to perform five feature tests to get the inference result of 7,890. This inference result is then compared with the comparison value of the DTP, i.e., 30,000, to determine that Record ❶ does not satisfy the predicate. However, since neither of the inference results on the leaf nodes of Node $md \leq 18,659$ satisfies the comparison of $> 30,000$, the feature test for Record ❶ on the internal Node $md \leq 18,659$ is clearly unnecessary. Furthermore, no matter whether a data record reaches the left child or the right child of Node $type \leq 1$, the final inference result will not exceed 30,000. Similarly, Node $size \leq 50,093$ also exhibits this characteristic. In summary, there are three redundant feature tests with this same pattern for evaluating the DTP with Record ❶. In general, these redundant feature tests for Record ❶ arise from the two children of internal nodes. Thus, we refer to this type of redundancy as sibling-induced redundant feature tests.

Ancestor-Induced Redundant Feature Test. Considering the decision path of Record ❸, i.e., $(store \leq 20) \rightarrow (dept \leq 15) \rightarrow (md \leq 862) \rightarrow (size \leq 17,112) \rightarrow (dept \leq 2) \rightarrow 20,581$, it requires five feature tests to obtain an inference result of 20,581. Then, this inference result is compared with the comparison value, i.e., 30,000, to decide that Record ❸ does not satisfy the predicate. However, there is another kind of redundant feature test in this case. Along the decision path, the feature *dept* is tested twice, forming the constraint $2 < dept \leq 15$. On the one hand, all records routed to the right subtree of Node $dept \leq 15$ already yield inference results below 30,000, leading the comparison result to be false. On the other hand, although Record ❸ is routed to the



(a) DTP in Example 1 after Predicate Rewrite



(b) DTP in Example 1 after Subtree Collapse

Fig. 2. Sibling-Centric Elimination for the Decision Tree Predicate in Example 1

left subtree of the same node, it also results in a false comparison result. Hence, the feature test at Node $dept \leq 15$ does not affect the final comparison results and can thus be considered redundant. Unlike the sibling-induced redundant feature test, this redundancy arises from Node $dept \leq 15$ and Node $dept \leq 2$ on the decision path, which constitutes an ancestor-descendant relation. Thus, we refer to this type of redundancy as ancestor-induced redundant feature tests.

High Frequency of Redundant Feature Tests. We evaluated the query of Example 1 in the Walmart dataset [14] and noticed that around 70% of the data records suffer from both types of redundant feature tests. Hence, eliminating these redundant feature tests offers substantial optimization potential. This motivates us to study a sibling-centric elimination method and an ancestor-centric elimination method to achieve this optimization target.

3 Sibling-centric elimination

We elaborate on the homogeneous sibling nodes in Section 3.1, and propose the merging-based subtree collapse method in Section 3.2 to enable the sibling-centric redundant feature test elimination.

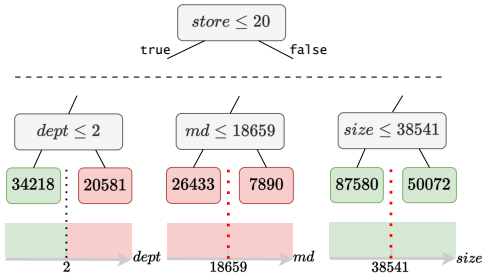


Fig. 3. Space Partitions of Three Nodes in the DTP from Example 1

3.1 Homogeneous Sibling Nodes

Following the idea that obtaining the final comparison result of DTP does not require the exact inference result from the model, we can first pass the comparison operator and comparison value into the tree model to make the model directly return the comparison result of the DTP. As depicted in Figure 2(a), the sibling-induced redundant feature tests for Example 1 are clearly exposed. Note that we use 1 to represent the comparison result of true and 0 to represent false. Starting from Node $dept \leq 15$, none of the leaves in its right subtree yield a true final comparison result. This pattern is similar to Node $size \leq 38,541$ and Node $type \leq 2$. Accordingly, we are able to conclude the definition of the homogeneous sibling nodes from the tree structure perspective.

Definition 1. *Homogeneous sibling nodes are a pair of sibling nodes where the subtrees rooted with either node have the same comparison result in the leaf nodes.*

For instance in Figure 2(a), the leaf children of Node $md \leq 18,659$, $size \leq 38,541$, and $type \leq 2$ are homogeneous sibling nodes. Recursively, the children of Node $type \leq 1$ and $size \leq 50,093$ are also homogeneous. Intuitively, these homogeneous sibling nodes lead to the redundant feature tests on their parent nodes.

Furthermore, to systematically characterize the pattern of redundant feature tests induced by such homogeneous sibling nodes, we build upon the space partition perspective used in prior works on decision trees [6, 22, 53]. Given an n-dimensional linear space $\mathbb{R}^n$, a hyperplane is an $(n-1)$-dimensional subspace of it, denoted as

$$H = \{\boldsymbol{x} \in \mathbb{R}^n \mid \boldsymbol{a}^T\boldsymbol{x} + b = 0, \quad \boldsymbol{a} \in \mathbb{R}^n, \quad b \in \mathbb{R}\}$$

where $\boldsymbol{x}$ is a point in the hyperplane and $\boldsymbol{a}$ is the normal vector of the hyperplane. A hyperplane divides an n-dimensional linear space into two parts. Each part is called a half-space, denoted as

$$S = \{\boldsymbol{x} \in \mathbb{R}^n \mid \boldsymbol{a}^T\boldsymbol{x} + b \leq 0, \quad \boldsymbol{a} \in \mathbb{R}^n, \quad b \in \mathbb{R}\}$$

where $\boldsymbol{x}$ is a point in the half-space.

Thereby, a decision tree with n features represents a partition of an n-dimensional linear space $\mathbb{R}^n$. Each internal node of the tree represents a half-space S, and its boundary is a hyperplane H.

$$S = \{\boldsymbol{x} \in \mathbb{R}^n \mid x_i + b \leq 0,\ b \in \mathbb{R},\ i \in \{1, \ldots, n\}\}$$

$$H = \{\boldsymbol{x} \in \mathbb{R}^n \mid x_i + b = 0,\ b \in \mathbb{R},\ i \in \{1, \ldots, n\}\}$$

where $\boldsymbol{x}$ is a data record and n is the number of its features.

Figure 3 illustrates the space partitions of three nodes in the DTP from Example 1. Each of them corresponds to a one-dimensional space. The green areas correspond to the green leaf nodes, indicating that the inference result of the model in these areas will lead the predicate to return true. The red areas correspond to the red leaf nodes, indicating that the inference result of the model in

these areas will lead the predicate to return false. Hence, from the perspective of space partition, the parent node of homogeneous sibling nodes represents a hyperplane that divides the space with both sides having the same comparison result. Such a hyperplane does not refine the partition and corresponds to a redundant feature test. Therefore, it is safe to merge the divided half-spaces to remove the parent node of homogeneous sibling nodes.

3.2 Merging-based Subtree Collapse

Based on the above description, we propose the merging-based subtree collapse method to achieve sibling-centric elimination. In particular, this method first rewrites the DTP and then applies subtree collapse by merging the partitioned half-spaces to remove all the homogeneous sibling nodes.

3.2.1 Predicate Rewrite. For a DTP that follows the format introduced in Section 2.1, we propagate the comparison operator `COMPARE` and comparison value `CONST` into the `PREDICTION` component. Each leaf node of the tree is then replaced with 1 or 0, indicating whether the DTP comparison result is true or false, respectively. This transformation converts the predicate into the following unified form: `PREDICTION' = 1`. Here, `PREDICTION'` denotes an inference function whose underlying tree model has leaf nodes labeled with either 0 or 1. The transformation is performed on the unified IR, which also serves as the query plan, as illustrated in Figure 1(b) for the query in Example 1. In this plan, the structure of the model is explicitly preserved. In Figure 1(b), the inference results corresponding to the red leaf nodes do not satisfy the predicate and are thus replaced with 0, whereas those corresponding to the green leaf nodes satisfy the predicate and are assigned 1. The rewritten predicate is shown in Figure 2(a), where the original tree model has been transformed to directly output the comparison results of the DTP. This rewrite also exposes pairs of homogeneous sibling nodes, enabling further subtree collapse.

3.2.2 Recursive Subtree Collapse. After predicate rewrite, we use the subtree collapse method to remove those homogeneous sibling nodes. As mentioned in Section 3.1, the subtree collapse process is conducted based on removing the hyperplane and merging the partitioned half-spaces if both the half-spaces lead to the same comparison result. In particular, this corresponds to collapsing a pair of homogeneous sibling nodes and their parent into a single node that preserves the original comparison result of the sibling nodes. Similar to the post-pruning [28] method used in the decision tree training phase, this process is recursively applied until there are no homogeneous sibling nodes left in the tree.

The detailed process of our merging-based subtree collapse after predicate rewrite is described in Algorithm 1. It is defined as a recursive function (Line 2) that traverses the tree in a post-order. First, we check whether the current traversing node of the function is a leaf node (Line 3). If so, we will directly return the comparison result of that leaf node. If not, the function will be recursively called for the left and right child nodes of the current traversing node (Line 5 and Line 6), and the return values are stored in $left\_value$ and $right\_value$, respectively. Then, we check if $left\_value$ and $right\_value$ are equal and not equal to $Null$ (Line 8). If not, we will return $Null$, indicating that there are no more homogeneous sibling nodes. If so, we will transform the node into a leaf node with the comparison result of $left\_value$ and return this result.

In Figure 2(a), collapsing the homogeneous sibling nodes whose parent is Node $md \leq 18,659$ reveals a new pair of homogeneous sibling nodes under Node $dept \leq 71$, enabling further subtree collapse. This process continues iteratively until no homogeneous sibling nodes remain, yielding the DTP shown in Figure 2(b). For the data records in Table 1, Record ❶ needs to take five feature tests with the DTP in Figure 2(a) to get the comparison result, while only two feature tests are needed after subtree collapse. Similarly, the number of feature tests for Record ❷ is reduced from five to four.

Algorithm 1 The Process of Subtree Collapse

Input: A decision tree predicate M after rewrite
Output: An optimized decision tree predicate

```
// Collapse subtrees of M by traversing from root node
function COLLAPSE_SUBTREE(node)
    if node is leaf then
        return node.value
    left_value ← COLLAPSE_SUBTREE(node.left)
    right_value ← COLLAPSE_SUBTREE(node.right)
    // Determine whether the sibling nodes are homogeneous
    if left_value = right_value and left_value ≠ Null then
        node.value ← left_value
        node.left ← Null
        node.right ← Null
        return node.value
    return Null
```

Evaluation Overhead Reduction. Recall the evaluation overhead of DTP in Equation (1), assume that the decision tree after subtree collapse has m leaf nodes, then we have $m \leq n$. Denote the evaluation overhead after the subtree collapse as C^*, then it holds that $C^* = \sum_{j=1}^{m} f_j^* c_j^* \leq \sum_{i=1}^{n} f_i c_i = C$. This indicates that the decision path for each data record is guaranteed to be equal to or shorter than that before subtree collapse. Furthermore, our sibling-centric elimination is a data-independent optimization that is effective for all data records in the table.

Impact of Selectivity. The occurrence of homogeneous sibling nodes is closely related to the selectivity of DTP: high selectivity produces more satisfying leaf nodes, whereas low selectivity results in more non-satisfying ones. Both scenarios will increase the likelihood of encountering homogeneous sibling nodes. In extreme cases, when the selectivity of a DTP is 0 or 1, the entire predicate can be replaced with a boolean constant.

4 Ancestor-centric Elimination

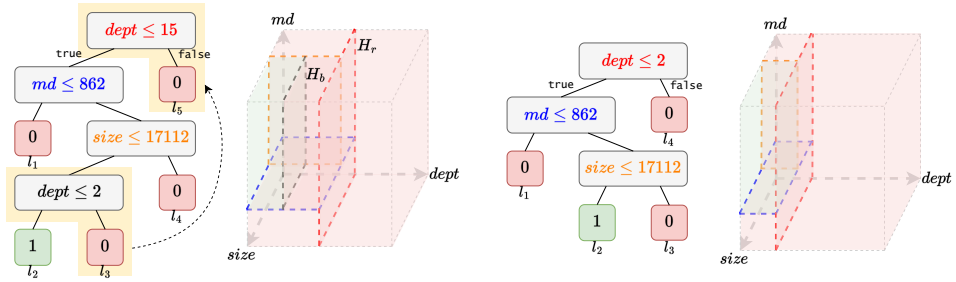
We introduce implicative ancestor nodes in Section 4.1, and propose the sliding-based subtree recombination method in Section 4.2 to achieve the ancestor-centric redundancy elimination.

4.1 Implicative Ancestor Nodes

After sibling-centric elimination, there are still ancestor-induced redundant feature tests left in the DTP. Considering Subtree I shown in Figure 2(b), the feature test for Record ❸ with $dept \leq 15$ is still redundant as described in Section 2.2. In fact, for data records that reach Subtree I, if $dept > 15$, the predicate will always return false. If $dept \leq 15$, the comparison result of the predicate is determined by the feature test at Node $dept \leq 2$. This pattern is similar to Node $dept \leq 95$ and Node $dept \leq 94$ in Subtree II. Accordingly, we are able to summarize the definition of the implicative ancestor nodes from the tree structure perspective.

Definition 2. *An implicative ancestor node is an internal ancestor node that tests the same feature as one of its descendant nodes, and their feature tests inherently have an implication relation.*

In contrast to the homogeneous sibling nodes, the redundant feature tests induced by such implicative ancestor nodes are not straightforward to discover. Therefore, we turn to the space partition discussed in Section 3.1 to systematically characterize the pattern of such redundancies. Considering Subtree I in Figure 2(b), its space partition is shown in Figure 4(a). Since subtree I has



(a) Space partition for Subtree I (b) Subtree I after Subtree Recombination

Fig. 4. Subtree Recombination for Subtree I

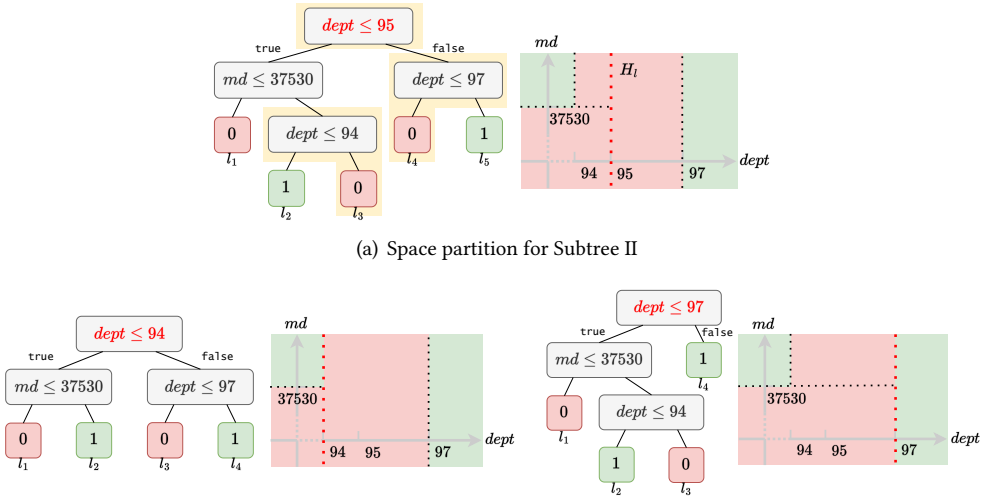
three different features, it represents a partition in a three-dimensional space. The hyperplanes are represented by dotted lines, and their colors correspond one-to-one to the colors of the text in the internal nodes of the tree. Node $dept \leq 15$ corresponds to the hyperplane represented by the red dotted lines H_r, and Node $dept \leq 2$ corresponds to the hyperplane represented by the black dotted lines H_b. Clearly, the subspace between H_r and H_b, as well as the subspace to the right of H_r, are both in red, which indicates the same comparison result for the DTP. Hence, the feature test at Node $dept \leq 15$ is redundant since H_r does not distinguish between these two subspaces. However, we cannot apply the approach described in Section 3.2 to merge the half-spaces divided by H_r, since those half-spaces are not entirely homogeneous. This motivates us to propose a novel sliding-based subtree recombination approach to conduct the ancestor-centric elimination.

4.2 Sliding-based Subtree Recombination

Given an internal node of a decision tree, we denote its corresponding half-space as P and its boundary hyperplane as H. If the comparison results remain identical in the areas adjacent to both sides of H, then H is called a *slidable hyperplane*. A slidable hyperplane can move along its normal vector or in the opposite direction until the comparison results on its two sides start to differ. This movable range is referred to as the *sliding interval*.

Given two internal nodes, there are three possible geometric relationships for their corresponding hyperplanes: they are orthogonal if they test different features, parallel if they test the same feature with different thresholds, and coincident if both the feature and threshold are identical. Our core idea is based on a key property of the slidable hyperplane: *an endpoint of a sliding interval always corresponds to a parallel hyperplane. When the slidable hyperplane reaches this endpoint, the two hyperplanes become coincident, which corresponds in the tree to merging two branches.* We refer to the two participating branches as the *sliding branch* and the *boundary branch*. The sliding branch is rooted at the *sliding node* representing the slidable hyperplane, and contains either its left or right subtree. The boundary branch consists of the *boundary node*, i.e., the node representing the hyperplane at the endpoint, and one of its leaf children. The sliding node is always an ancestor of the boundary node. When the slidable hyperplane slides to coincide with the boundary hyperplane, the boundary branch is merged into the sliding branch, and the threshold of the sliding node is replaced by that of the boundary node. The remaining child of the boundary node is reattached to the original parent of the boundary node, becoming the *risen node* and roots the *risen subtree*.

As illustrated in Figure 4(a), the hyperplane H_r can slide in the direction opposite to the $dept$ axis until it reaches plane H_b, where the comparison result begins to change. At this point, H_r coincides with H_b, so we turn them into one single plane. In this example, Node $dept \leq 15$ serves as



(a) Space partition for Subtree II

(b) Recombination Variant ① of Subtree II

(c) Recombination Variant ② of Subtree II

Fig. 5. Subtree Recombination for Subtree II

the sliding node, and the yellow region with it is the sliding branch. Node $dept \leq 2$ is the boundary node, and the yellow region with it constitutes the boundary branch. The leaf node l_2 becomes the risen node. Following the sliding process, the tree is recombined into a new structure, as shown in Figure 4(b), effectively eliminating the redundant feature test at Node $dept \leq 15$ through the sliding-based recombination.

4.2.1 The Dilemma of Sliding. Although the sliding approach described above is able to eliminate the redundant feature tests induced by implicative ancestor nodes, the tree recombination following sliding may increase the number of feature tests required for certain data records. To analyze this potential side effect, we first categorize slidable hyperplanes into three types based on the boundedness of their sliding intervals.

- Infinite slidable hyperplane, if its sliding interval is infinite in both directions, as illustrated by the red dotted lines in Figure 3. In this case, sliding only changes the threshold of the sliding node and does not change the tree structure. Thus, sliding has no effect for redundant feature test elimination. In fact, this is the case described in Section 3.1, and we can leverage the merging-based subtree collapse approach to resolve it.
- Semi-infinite slidable hyperplane, if its sliding interval is infinite in one of the directions, as illustrated by the red plane H_r in Figure 4(a). In this case, we have two options, i.e., not sliding the hyperplane or sliding it to the endpoint of its sliding interval. Each option corresponds to a form of recombination.
- Finite slidable hyperplane, if its sliding interval is finite in both directions, as illustrated by the red dotted line H_l in Figure 5(a). In this case, we have three options, i.e., not sliding, sliding to the left endpoint of the interval, or sliding to the right endpoint. Each option corresponds to a form of recombination.

Then, let us consider Subtree II shown in Figure 5(a), the sliding interval of its root node has two endpoints, which means there are two possible sliding directions, resulting in two recombination variants. When the root node slides to the left endpoint, Subtree II is recombined into Variant ①

in Figure 5(b). On the contrary, when the root node slides to the right endpoint, Subtree II is transformed into Variant ② in Figure 5(c). For Record ❺ in Table 1, it only needs to undergo two feature tests in both Subtree II and Variant ①. However, Record ❺ needs to perform three feature tests in Variant ②, which even brings one additional feature test. This motivates us to seek an approach that eliminates the ancestor-induced redundant feature tests without side effects for all data records.

4.2.2 Recursive Subtree Recombination. To achieve the aforementioned goal, we rely on the following heuristic: it is preferable to move a slidable hyperplane toward the side with a more complex space partition, as this shifts more data records into the other side with a simpler partition, thereby shortening their decision paths. Based on this heuristic, we propose the recursive subtree recombination approach, which quantifies space partition complexity via evaluation overhead and recursively considers sliding options between the sliding node and its descendant nodes. We refer to the subtree rooted at the sliding node as the *sliding subtree*, and analyze the change in evaluation overhead to guide the sliding choices.

We divide the set J, which consists of the indices of all leaf nodes in the tree obtained after subtree collapse, into the following five disjoint subsets based on the position of the node.

$$\begin{aligned}
J_n &= \{j \in J \mid l_j \text{ is not in the sliding subtree}\} \\
J_s &= \{j \in J \mid l_j \text{ is in the sliding branch}\} \\
J_b &= \{j \in J \mid l_j \text{ is in the boundary branch}\} \\
J_r &= \{j \in J \mid l_j \text{ is in the risen subtree}\} \\
J_o &= J - J_n - J_s - J_b - J_r
\end{aligned}$$

Assume that after subtree recombination, the tree retains s leaf nodes. The evaluation overhead of the DTP is then given by $C' = \sum_{k=1}^{s} f'_k c'_k$. Let $K = \{1, 2, \ldots, s\}$ represent the set of indices for the leaf nodes. Similar to J, we divide K into five subsets: K_n, K_s, K_b, K_r, and K_o. It is worth noting that, due to the combination of the sliding branch and the boundary branch, we have $K_s = K_b$. Furthermore, we divide the evaluation overhead of the DTP according to the division of the leaf nodes. The evaluation overhead of the four parts after subtree recombination is denoted as C'_n, C'_s, C'_r, and C'_o, respectively. Except for the nodes that are not in the sliding subtree, i.e., the leaf nodes with indices in J_n, all other nodes are affected by the subtree recombination. In general, after subtree recombination, for the data records that reach the sliding node, the frequency of entering the sliding branch increases, while that of entering the other branch decreases. For a leaf node with index $j \in J_r$ or $j \in J_o$, denote the frequency reduction as Δf_j^*. Let $t^*(\boldsymbol{x})$ and $t'(\boldsymbol{x})$ denote the leaf indices reached by record $\boldsymbol{x}$ before and after subtree recombination, respectively. Further, define $X(j, A, B) = \{\boldsymbol{x} \mid t^*(\boldsymbol{x}) = j \in A \subseteq J, t'(\boldsymbol{x}) \in B \subseteq K\}$, then the change in total overhead can be expressed as

$$\begin{aligned}
C' - C^* &= C'_n + C'_r + C'_o + C'_s - C^* \\
&= \sum_{j \in J_r} (\Delta f_j^* (\frac{\sum_{\boldsymbol{x} \in X(j, J_r, K_s)} c'_{t'(\boldsymbol{x})}}{|X(j, J_r, K_s)|} - c_j^*) + \Delta f_j^* - f_j^*) \\
&+ \sum_{j \in J_b} f_j^* (\frac{\sum_{\boldsymbol{x} \in X(j, J_b, K_s)} c'_{t'(\boldsymbol{x})}}{|X(j, J_b, K_s)|} - c_j^*) \\
&+ \sum_{j \in J_o} \Delta f_j^* (\frac{\sum_{\boldsymbol{x} \in X(j, J_o, K_s)} c'_{t'(\boldsymbol{x})}}{|X(j, J_o, K_s)|} - c_j^*)
\end{aligned} \tag{2}$$

By doing so, we have two theorems that ensure the evaluation overhead of the DTP does not increase during subtree recombination.

Algorithm 2 The Process of Subtree Recombination

Input: A decision tree predicate M after subtree collapse
Output: An optimized decision tree predicate

```
// Recombine subtrees of M by traversing from root node
function RECOMBINE_SUBTREE(node)
    if node is leaf then return
    RECOMBINE_SUBTREE(node.left)
    RECOMBINE_SUBTREE(node.right)
    H ← node.to_hyperplane()
    space ← the space partition of the subtree rooted at node
    slidable ← whether H is a slidable hyperplane in space
    if slidable = true and H.endpoints.size() = 1 then
        slide H to H.endpoint return
    if slidable = true and H.endpoints.size() = 2 then
        // Determine the sliding direction by overhead scaling
        c*_min ← the shortest path length in node.left_subtree
        c'_max ← the longest path length in node.right_subtree
        if c'_max − c*_min ≤ 0 then
            slide H to H.left_endpoint return
        c*_min ← the shortest path length in node.right_subtree
        c'_max ← the longest path length in node.left_subtree
        if c'_max − c*_min ≤ 0 then
            slide H to H.right_endpoint
```

Theorem 1. *For one semi-infinite slidable hyperplane, sliding it to the endpoint of its sliding interval does not increase the overhead.*

Proof. For one semi-infinite sliding node, the sliding branch contains only two nodes: one is the sliding node itself, and the other is a leaf node with an inference result identical to that of the leaf node in the boundary branch. Denoting the number of feature tests required to reach the sliding node as α, for one data record $\boldsymbol{x}$ with $t'(\boldsymbol{x}) \in K_s$, we have $c'_{t'(x)} = \alpha + 1$. Moreover, for any leaf nodes in the sliding subtree, i.e., $j \in J - J_n$, it holds that $\alpha + 1 \leq c^*_j$ and $\Delta f^*_j \leq f^*_j$. Therefore, according to Equation (2), we have

$$\begin{aligned} C' - C^* = &\sum_{j \in J_r} (\Delta f^*_j (\alpha + 1 - c^*_j) + \Delta f^*_j - f^*_j) \\ &+ \sum_{j \in J_b} f^*_j (\alpha + 1 - c^*_j) + \sum_{j \in J_o} \Delta f^*_j (\alpha + 1 - c^*_j) \leq 0 \end{aligned} \tag{3}$$

This means that, for one semi-infinite sliding node, subtree recombination with sliding does not increase the overhead. □

For one finite sliding node, according to Equation (2), comparing C' and C^* needs the number of feature tests required for the data records passing through the sliding node, both before and after subtree recombination. However, this information is unavailable unless we evaluate all data records against each of the three DTPs individually. Consequently, we cannot identify the option with the lowest overhead among the three. Fortunately, we can determine whether a slide reduces the evaluation overhead by scaling Equation (2).

Let $X_s = \{\boldsymbol{x} | t^*(\boldsymbol{x}) \notin J_s, t'(\boldsymbol{x}) \in K_s\}$ denote the set of data records whose decision paths are altered due to the subtree recombination. Also, let $c'_{max} = \max_{\boldsymbol{x} \in X_s} c'_{t'(\boldsymbol{x})}$ denote the maximum number of feature tests required after sliding among all data records in X_s, and let $c^*_{min} = \min_{\boldsymbol{x} \in X_s} c^*_{t^*(\boldsymbol{x})}$ denote the minimum number of feature tests required before sliding among all data records in X_s.

Theorem 2. *For one finite slidable hyperplane, if there is a sliding option that satisfies $c'_{max} \leq c^*_{min}$, then the subtree recombination will not increase the overhead.*

Proof. When $c'_{max} \leq c^*_{min}$, we have

$$\begin{aligned} C' - C^* \leq & \sum_{j \in J_r} (\Delta f^*_j (c'_{max} - c^*_{min}) + \Delta f^*_j - f^*_j) \\ & + \sum_{j \in J_b} f^*_j (c'_{max} - c^*_{min}) + \sum_{j \in J_o} \Delta f^*_j (c'_{max} - c^*_{min}) \leq 0 \end{aligned} \tag{4}$$

Thus, the evaluation overhead is guaranteed not to increase. □

Considering the subtree recombination from Subtree II to Variant ① shown in Figure 5(b), if a data record reaches leaf node l_1 or l_3 of Subtree II before subtree recombination, then it may reach l_3 of Variant ① afterward. Thus, $c'_{max} = \max\{2\} = 2$, $c^*_{min} = \min\{2, 3\} = 2$, $c'_{max} = c^*_{min}$, and this subtree recombination will not increase the overhead. Similarly, considering the subtree recombination from Subtree II to Variant ② shown in Figure 5(c), if a data record reaches l_4 of Subtree II before subtree recombination, then it may reach l_1 or l_3 of Variant ② afterward. Thus, $c'_{max} = \max\{2, 3\} = 3$, $c^*_{min} = \min\{2\} = 2$, $c'_{max} > c^*_{min}$, and it is uncertain whether this subtree recombination will increase the overhead. Therefore, we will choose the first option.

The detailed process of our subtree recombination method is described in Algorithm 2. Similar to Algorithm 1, the whole process is a recursive function that traverses the tree in post-order (Line 2). First, we check whether the input node of the function is a leaf node. If so, the function returns directly. If not, the function is recursively called for the left and right child nodes of the input node. For each internal node, we first determine whether its corresponding hyperplane H is a slidable hyperplane by space partition, and store the result in a boolean variable *slidable* (Line 6 – Line 8). If *slidable* is true and the sliding interval of H has one endpoint, H is a semi-infinite slidable hyperplane. In this case, we slide H to the endpoint to recombine the subtree according to Theorem 1 (Line 10). If *slidable* is true and the sliding interval of H has two endpoints, H is a finite slidable hyperplane. We then determine the sliding direction by overhead scaling and slide H to the selected endpoint according to Theorem 2 (Line 13 – Line 20).

The DTP of Example 1 after subtree recombination is depicted in Figure 6. For Record ❸ in Table 1, it needs to take five feature tests with the DTP in Figure 2(b), while it only needs to perform the feature test twice with the DTP in Figure 6. Similarly, the number of feature tests for Record ❹ is reduced from four to two.

Evaluation Overhead Reduction. According to Theorem 1 and Theorem 2, our sliding-based subtree recombination approach does not increase the DTP evaluation overhead, i.e., $C^* \leq C'$.

5 Random Forest Predicate

In this section, we extend optimizations discussed in Section 3 and 4 to random forest predicates. Assume that a random forest contains t decision trees. If it is a regressor, for one data record, the inference result of the i-th decision tree is denoted as v_i, then the inference result of the random forest is the average of the inference results of all decision trees, i.e., $v = \frac{1}{t} \sum_{i=1}^{t} v_i$, which is called *averaging*. If the random forest is a binary classifier, for one data record, the positive-class probability predicted by the i-th decision tree is denoted as p_i, and the inference result of the decision tree is $v_i = \mathbb{I}(p_i > 0.5)$, where $\mathbb{I}$ is the indicator function. Then, there are two ways to

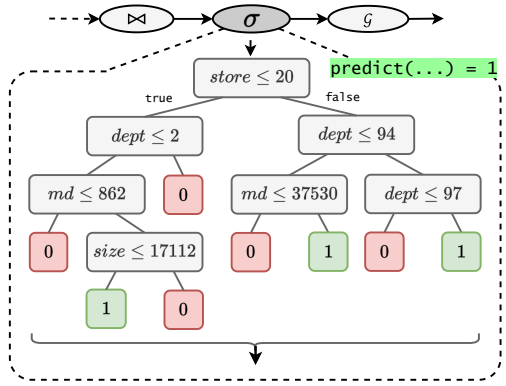


Fig. 6. DTP in Example 1 after Subtree Recombination

calculate the inference result of the random forest. One is called *hard voting*, which takes the majority vote of the individual tree inference results, i.e., $v = \mathbb{I}(\frac{1}{t}\sum_{i=1}^{t} v_i > 0.5)$. The other is called *soft voting*, which first averages the predicted probabilities across all trees, i.e., $p = \frac{1}{t}\sum_{i=1}^{t} p_i$, then $v = \mathbb{I}(p > 0.5)$.

Accordingly, we apply sibling-centric elimination and ancestor-centric elimination to each decision tree individually since each decision tree in the random forest contributes equally. Similarly, we first pass the comparison operator and comparison value into the model through predicate rewrite. After this rewrite, regardless of whether the original tree is a regressor or a classifier, each decision tree becomes a binary classifier that outputs 0 or 1. Therefore, we use a hard-voting-like method to calculate the comparison result of the random forest predicate. Since this approach replaces averaging and soft voting with hard voting, it may introduce deviation in predicate accuracy, i.e., the accuracy of the model in predicting whether the predicates are satisfied. In detail, using $\mathbb{F} : \mathbb{R} \to \{0, 1\}$ to represent the binary-valued function of the predicate applied to the model inference result, for a random forest regressor predicate, the comparison result is changed from $\mathbb{F}(v) = \mathbb{F}(\frac{1}{t}\sum_{i=1}^{t} v_i)$ to $\mathbb{I}(\frac{1}{t}\sum_{i=1}^{t} \mathbb{F}(v_i) > 0.5)$. This transformation introduces a deviation due to the nonlinearity of $\mathbb{F}$, since $\mathbb{F}(\frac{1}{t}\sum_{i=1}^{t} v_i) \neq \frac{1}{t}\sum_{i=1}^{t} \mathbb{F}(v_i)$. Similarly, for a random forest binary classifier predicate, denoted as $\mathbb{F}'(x) = \mathbb{F}(\mathbb{I}(x > 0.5))$, the comparison result is changed from $\mathbb{F}(v) = \mathbb{F}'(\frac{1}{t}\sum_{i=1}^{t} p_i)$ to $\mathbb{I}(\frac{1}{t}\sum_{i=1}^{t} \mathbb{F}'(p_i) > 0.5)$.

Due to the nonlinearity and jump discontinuities of $\mathbb{F}$ and $\mathbb{F}'$, deriving a rigorous bound on the accuracy deviation is difficult. However, we can characterize the factors that influence this deviation. It is affected by (1) the variance among the inference results of individual trees in a regressor, or the variance among the positive-class probabilities predicted by individual trees in a binary classifier, denoted as *inter-tree variance*, and (2) the distance between the regressor inference result and the jump discontinuities of $\mathbb{F}$, or the distance between the positive-class probability predicted by the classifier and the jump discontinuities of $\mathbb{F}'$, denoted as *discontinuity distance*. In general, a smaller inter-tree variance and a greater discontinuity distance reduce the deviation, whereas the opposite conditions amplify it. For a well-trained random forest, inter-tree variance tends not to be large on most data records, as all constituent trees are trained to approximate the same underlying ground truth. Moreover, our experiments in Section 7.4 show that the resulting deviation in predicate accuracy is minor in practice.

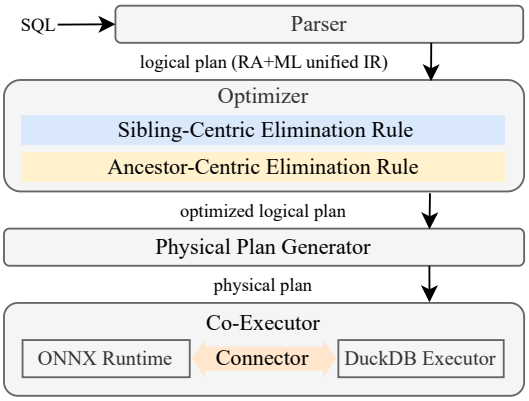


Fig. 7. System Architecture of ReTree

6 System Implementation

We implement ReTree on DuckDB with ONNX Runtime, since both of them have high performance and scalability [27, 38]. Based on the framework of DuckDB extensions, we implemented ReTree as a flexible and pluggable extension of DuckDB.

Figure 7 shows the complete execution flow of a query in SQL with a decision tree or random forest predicate. The ML model in the predicate is wrapped in a `predict` function expression. This function acts as a connector between ONNX Runtime and DuckDB, taking a model path and features as input and returning the corresponding inference result. First, the parser parses the SQL statement into a logical plan, which is a unified IR of RA and ML operators. Then, the optimizer optimizes the logical plan with both sibling-centric elimination and ancestor-centric elimination methods. Next, the optimized logical plan is further converted into a physical plan and given to the co-executor for execution. The relational algebra part is executed by the DuckDB executor, and the model inference part is executed by ONNX Runtime.

Portability ReTree is not tied to DuckDB and can be readily extended to other databases by implementing an ONNX Runtime connector for ML inference and adding two logical optimization rules for sibling-centric and ancestor-centric elimination.

7 Experimental Studies

We first present the experimental setting in Section 7.1. Then, we evaluate the efficiency of ReTree in Section 7.2 and Section 7.3. Next, we discuss the accuracy of random forest predicates in Section 7.4. We finally make a further discussion in Section 7.5.

7.1 Experimental Setting

We conduct all experiments on a server machine equipped with Intel(R) Xeon(R) Gold 6240R CPU @ 2.40GHz (48 physical cores) with 1.5 MB, 48 MB, and 71.5 MB L1, L2, and L3 caches and 768 GB DDR4 RAM. The software stack includes Ubuntu 20.04, Python 3.10, DuckDB 1.2.1, and ONNX Runtime 1.20.1. By default, for the decision tree predicate and random forest predicate experiments, we set the database parallelism to 1 and 4, respectively.

Datasets. We use two kinds of datasets in our experimental studies. First, we employ publicly available datasets, including Walmart, NYC Taxi, Medical Charges, Flights, and Bike Sharing, which are widely used in data science tasks [8, 14, 17]. Additionally, we use benchmark datasets from TPCx-AI [3] and TPC-H to assess the optimization performance and usability in typical industrial

Table 2. Experimental Prediction Queries

ID	Use Case	Dataset	#feature	Predicate
Q1	Amounts Prediction	NYC Taxi	9	DT
Q2	Charges Forecast	Medical Charges	3	DT
Q3	Trips Classification	TPCx-AI	77	DT
Q4	Sales Forecast	Walmart	16	RF
Q5	Codeshare Prediction	Flights	20	RF
Q6	Amounts Prediction	NYC Taxi	9	RF
Q7	Usage Forecast	Bike Sharing	16	RF
Q8	Profit Measure	TPC-H	4	RF

```
SELECT <Projection1>, <Projection2>, ...
FROM table1 t1 JOIN table2 t2 ON t1.a = t2.a
JOIN table3 t3 ON t2.a = t3.a JOIN ...
WHERE <MLPredicate> AND <Predicate1> AND ...;
```

Fig. 8. SPJ-based Query Template

scenarios. All datasets include both numerical and categorical columns, with the number of features after encoding listed as #feature in Table 2. To match production-scale dataset sizes, we scale the publicly available datasets to 10GB (after joining) by replicating each dataset several folds, following the experiments in Raven [15, 31]. For the TPC toolkits, we apply a default scale factor of 10 (SF=10).

Trained Pipelines. As summarized in Table 2, we evaluate ReTree against decision trees (DT) and random forests (RF). Each trained pipeline includes a featurizer for categorical inputs, which are encoded using one-hot encoding. Among the pipelines, Q3 and Q5 implement classification tasks: Q3 is binary and Q5 is multi-class, while all others correspond to regression tasks. All pipelines are trained using scikit-learn with default hyperparameters and then converted to the ONNX format. Both the decision tree and random forest models used in the experiments have a maximum depth of 10, with the random forest consisting of 100 estimators.

Prediction Queries. To ensure representativeness, we adapt our prediction queries from the methodologies and workload patterns of established works such as Smart [10], Raven [15, 31], and IMBridge [57, 58]. We construct three types of queries as follows:

- *SPJ-based Queries (Q1, Q2, Q4-Q7)*: These queries are constructed over publicly available datasets following the classic and widely adopted Select–Project–Join (SPJ) SQL templates shown in Figure 8. Each query includes a decision tree or random forest predicate in the `WHERE` clause. We control the predicate selectivity to be 1%–5% for regression tasks, while using naturally low-selectivity categories for classification tasks. Queries for *NYC Taxi*, *Medical Charges*, and *Bike Sharing* involve single-table scans, whereas those for *Walmart* and *Flights* require three-way and four-way joins, respectively. For *Walmart* and *Bike Sharing*, specifically, we incorporate SQL-based data preprocessing steps that reproduce the feature engineering workflows of top-voted Kaggle notebooks, ensuring fidelity to real-world analytical pipelines.
- *TPCx-AI-based Query (Q3)*: We rewrite the original pandas-based Python codes of the UC08-Classification of Trips in TPCx-AI to an equivalent SQL-based query. The inference function is then encapsulated as a decision tree predicate in the `WHERE` clause.
- *TPC-H-based Query (Q8)*: This query is derived from the Product Type Profit Measure query in TPC-H. We modify the original analytical logic by replacing the profit calculation with a random forest predicate injected into the `WHERE` clause.

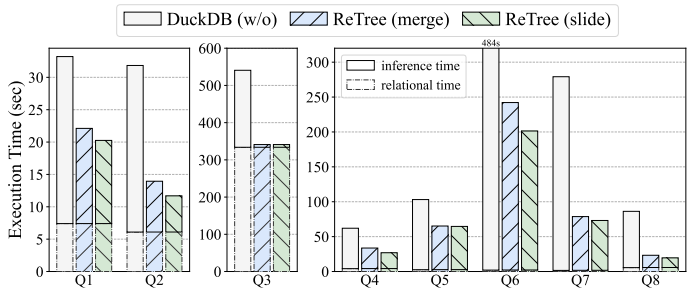


Fig. 9. End-to-End Execution Time

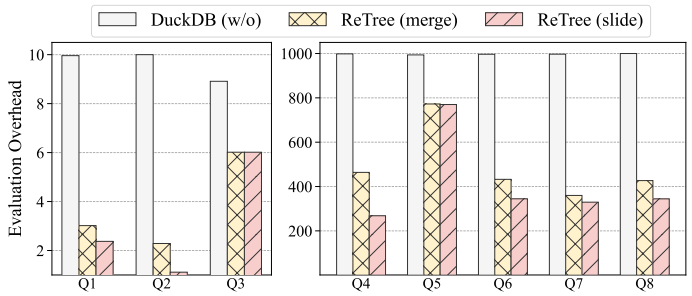


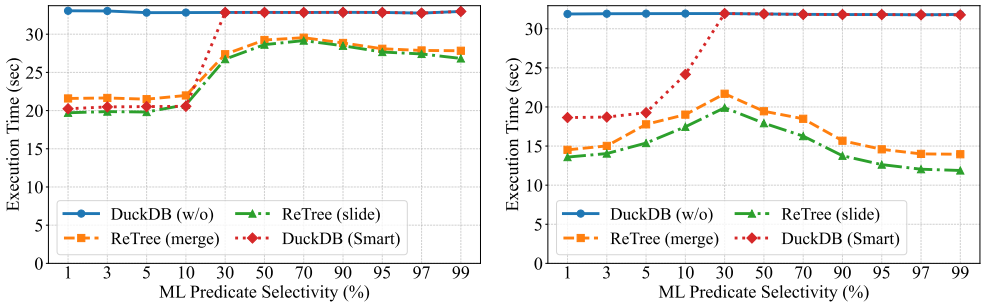
Fig. 10. Effect on Evaluation Overhead

7.2 Efficiency of Sibling-Centric Elimination

In this section, we evaluate the merging-based subtree collapse method in terms of performance and effectiveness.

7.2.1 End-to-End Execution Time. To evaluate the efficiency of the merging-based subtree collapse method, we measure the end-to-end execution time of ReTree without optimization as the baseline, denoted as DuckDB (w/o), and compare it to the execution time of ReTree with the merging-based subtree collapse method, denoted as ReTree (merge). We measure the execution time of the model inference and relational algebra processing, denoted as inference time and relational time, respectively. In particular, the relational time excludes the optimization overhead. We will further provide a detailed analysis of the optimization overhead in Section 7.5.3. As depicted in Figure 9, the subtree collapse method yields an average improvement of 2.26x compared to the baseline. For Q3, the overall speedup is relatively modest at 1.59x. This is attributed to the complexity of the relational algebra part, which shifts the performance bottleneck away from the model inference. Notably, when considering the inference time for Q3 in isolation, the improvement reaches 28.29x, highlighting the efficiency of the subtree collapse method. In contrast, while inference time dominates the execution of Q5, its overall speedup is only 1.58x. This stems from the fact that Q5 is a binary classification task where each split of the model is trained to distinguish between two classes, resulting in highly discriminative and inherently non-redundant decision paths with few homogeneous sibling nodes. In summary, the subtree collapse method improves query performance across various workloads.

7.2.2 Effect on Evaluation Overhead. To measure the effectiveness of the subtree collapse method in reducing evaluation overhead, we demonstrate the DTP evaluation overhead defined in Equation 1, as shown in Figure 10. The evaluation overhead of a random forest predicate is defined as the



(a) Selectivity Variation for Q1

(b) Selectivity Variation for Q2

Fig. 11. Impact of Selectivity

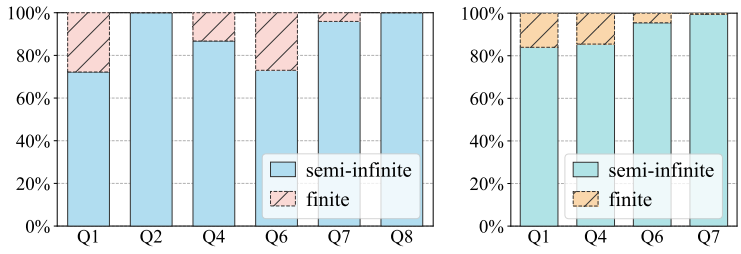
sum of the evaluation overhead of all individual trees. In detail, the average evaluation overhead decreases by 54% after optimization. Moreover, the extent of evaluation overhead reduction closely corresponds to the observed model inference speedup across queries, indicating a strong correlation between redundant feature test elimination and runtime performance improvement. Specifically, for queries with more significant optimization gains, such as Q2, Q4, and Q6–Q8, the evaluation overhead reduction is particularly notable, ranging from 54% to 77%. In contrast, for Q5, the overhead is reduced by only 22%, due to fewer redundant feature tests existing in the binary classifier. Hence, the subtree collapse method improves the prediction query execution by reducing the evaluation overhead of DTP.

7.2.3 Impact of Selectivity. Since the selection operation is a critical factor in determining the extent of homogeneous sibling nodes in decision tree and random forest predicates, we next examine the impact of ML predicate selectivity through two typical prediction queries, Q1 and Q2. The results are shown in Figure 11. As selectivity increases from 1% to 99%, the execution time of queries optimized by the subtree collapse method initially increases, then decreases. However, even at its peak, the subtree collapse method still outperforms the original DuckDB. Smart [10] derives simple comparison predicates from ML predicates and embeds them into queries to eliminate irrelevant tuples early. We implement it in DuckDB for comparison with ReTree, denoted as DuckDB (Smart). We find that Smart is not suitable for high-selectivity scenarios and fails to deliver performance benefits when the ML predicate selectivity exceeds 30%. In contrast, ReTree is faster than Smart even in low-selectivity scenarios. In summary, the results show that although the subtree collapse method is influenced by selectivity, it consistently improves performance, particularly for the ML predicates with extreme selectivity.

7.3 Efficiency of Ancestor-Centric Elimination

In this section, we study the performance and effectiveness of the sliding-based subtree recombination approach.

7.3.1 End-to-End Execution Time. To demonstrate the efficiency of ancestor-centric elimination, we measure the end-to-end execution time of the sliding-based subtree recombination approach, denoted as ReTree (slide). As shown in Figure 9, the subtree collapse method already achieves significant reductions in inference time, especially for Q3 and Q8. As a result, there is little room left for further optimization. Despite this, ReTree (slide) still delivers an average speedup of 1.15x over the subtree collapse method. Notably, the effectiveness of subtree recombination depends on the



(a) Quantity (b) Performance Improvement

Fig. 12. Slidable Hyperplane Ratio

structural simplifications enabled by subtree collapse. That explains why the subtree recombination approach has no impact on Q5, as it is a binary classification task with fewer feature tests that the subtree collapse method can eliminate. Moreover, ReTree (slide) demonstrates significantly improved performance over the original DuckDB, with an average 2.56x speedup in execution time. This performance improvement highlights the effectiveness of our proposed approach in eliminating redundant feature tests. In summary, subtree recombination accelerates query execution by further eliminating the ancestor-induced redundant feature tests.

7.3.2 Effect on Evaluation Overhead. Similar to Section 7.2.2, we also measure the DTP evaluation overhead after applying the sliding-based subtree recombination approach. As shown in Figure 10, the average evaluation overhead further decreases by an additional 20% compared to Retree (merge). However, for Q3 and Q5, the overhead remains almost unchanged, which explains the limited improvement in their end-to-end execution time. Overall, these results further confirm the effectiveness of the sliding-based subtree recombination approach in eliminating redundant feature tests.

7.3.3 Impact of Slidable Hyperplane Types. To demonstrate the effectiveness of different types of slidable hyperplanes in our subtree recombination approach, we measure the frequency of occurrence and the ratio of performance improvement of the finite slidable hyperplane, denoted as finite, and the semi-infinite slidable hyperplane, denoted as semi-infinite, on queries for which the sliding-based subtree recombination provides significant benefits. As shown in Figure 12(a), semi-infinite slidable hyperplanes overwhelmingly dominate among all slidable hyperplanes, comprising as much as 88% on average. In particular, finite slidable hyperplanes do not appear at all in Q2 and Q8. To further investigate the impact of different types of hyperplanes on performance improvement, we focus on those queries that do contain finite slidable hyperplanes. As illustrated in Figure 12(b), the contribution of finite slidable hyperplanes to overall execution performance is limited, accounting for an average of only 9%. This observation is consistent with our expectation, given their relatively low occurrence frequency across the workload. In conclusion, the results indicate that the semi-infinite slidable hyperplane contributes more substantially to both occurrence frequency and performance improvement compared to finite slidable hyperplanes. This highlights the effectiveness of our proposed method in leveraging semi-infinite slidable hyperplanes to enhance performance without side effects.

7.4 Accuracy of Random Forest Predicate

We focus on the deviation in predicate accuracy caused by the method described in Section 5, which extends decision tree predicates to random forest predicates. To evaluate this deviation,

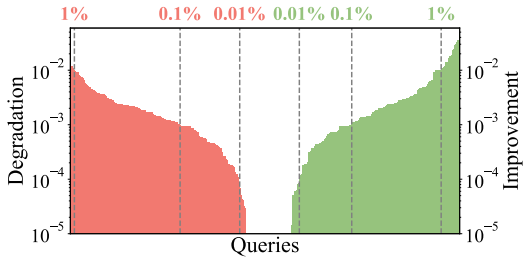


Fig. 13. Deviation in Accuracy of 442 Queries

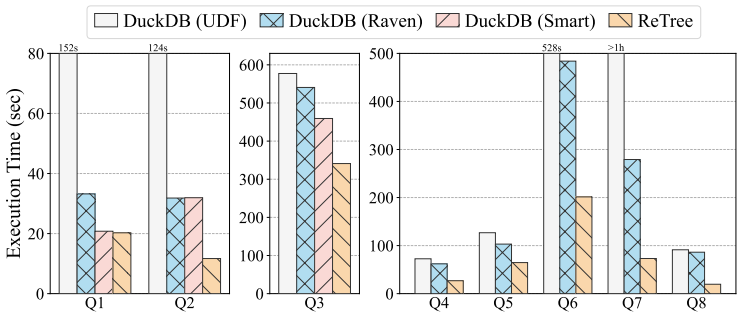


Fig. 14. Comparison with Other Solutions

we measure the accuracy of random forest predicates before and after applying the method on the tabular benchmark [8], a collection of 59 datasets comprising 23 binary classification and 36 regression tasks. Each dataset-model pair exhibits a distinct level of inter-tree variance. Since our focus is solely on the accuracy of the ML predicate, we adopt classic SPJ queries following the template shown in Figure 8, where each contains an ML predicate in the `WHERE` clause.

To assess the impact of discontinuity distance, we evaluate each dataset under varying comparison values. For regression tasks, we generate predicates with 11 different selectivities ranging from 1% to 99%. For classification tasks, we consider predicates corresponding to all class labels. This yields a total of 442 queries, enabling a comprehensive evaluation of predicate accuracy deviation across diverse scenarios. The experimental results are shown in Figure 13, where red regions indicate a degradation in predicate accuracy and green regions indicate an improvement. Among the 442 queries, predicate accuracy decreases for 45.2% of the queries, remains unchanged for 11.4%, and improves for 43.4%. The 99th, 95th, and 90th percentiles (p99, p95, and p90) of predicate accuracy deviation are 0.032, 0.011, and 0.007, respectively. Overall, these results demonstrate that ReTree preserves predicate accuracy with a minor deviation from the baseline.

7.5 Discussion

We make further discussions on the comparison with other solutions, the scalability, and the limitation of ReTree.

7.5.1 Comparison with Other Solutions. We showcase the advantages of ReTree in comparison to alternative solutions: Smart and Raven. Smart [10] generates simple comparison expression predicates based on ML predicates and integrates them into queries to filter out irrelevant tuples early. Raven [15, 31] simplifies ML predicates by leveraging other predicates that share the same

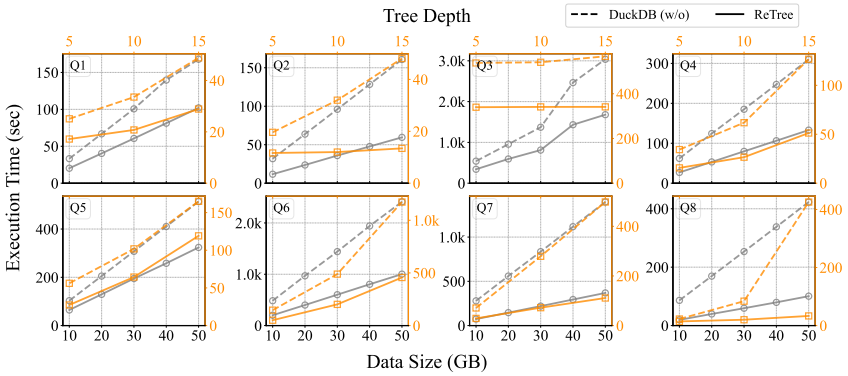


Fig. 15. Scalability

input columns to prune unnecessary tree branches. Note that we extend them to DuckDB and refer to them as DuckDB (Smart) and DuckDB (Raven), respectively, since both works are not open-source and their optimization techniques are database-independent. In addition, we compare the execution of prediction queries via Python UDF [7], denoted as DuckDB (UDF), which is a common approach for in-database prediction query execution. As shown in Figure 14, we present the overall performance across all prediction queries. Specifically, ReTree achieves an average speedup of 4.25x over Python UDFs, because UDF-based execution treats tree models as black boxes, preventing cross-optimization. Additionally, it incurs overhead from data conversion and copying. Compared to DuckDB (Raven), ReTree is 2.56x faster, as Raven is unaware of redundant feature tests caused by sibling nodes and ancestor nodes. Additionally, since DuckDB (Smart) only supports decision tree predicates, we exclude the experiments for Q4-Q8 on it. For Q1-Q3, ReTree delivers an average performance boost of 1.7x compared to DuckDB (Smart). This happens because Smart is more suitable for prediction queries with low selectivity and involving multiple table joins. To sum up, ReTree achieves superior performance compared to other solutions in optimizing ML predicates like Raven and Smart.

7.5.2 Scalability. We vary both data size and tree depth to evaluate ReTree's query performance on large-scale data and its scalability with respect to model complexity.

Data Scalability. For publicly available datasets, we scale the data from 10 GB to 50 GB. For benchmark datasets, we increase the scale factor from 10 to 50. As shown in Figure 15, ReTree consistently outperforms the original DuckDB for all dataset sizes, and demonstrates near-linear scalability as the data size increases, indicating that its performance scales proportionally with dataset growth.

Model Complexity. In Figure 15, we also examine the impact of ReTree's optimizations as tree depth increases. We evaluate models with maximum depths of 5, 10, and 15. ReTree consistently outperforms the original DuckDB across all tree depths. Although deeper trees result in exponentially larger models and super-linear growth in inference time, ReTree significantly reduces inference latency, particularly for Q8. Consequently, ReTree delivers performance improvements regardless of model complexity.

7.5.3 Optimization Overhead. To evaluate the trade-off between optimization overhead and execution speedup, we conduct experiments across varying model complexities and data scales (ranging from 1 MB to 1 GB). Specifically, we utilize Q1 for Decision Trees (varying depths of 3, 5, 10, and 15) and Q7 for Random Forests (fixed depth of 10, varying tree counts of 10, 30, 50, and 100) and

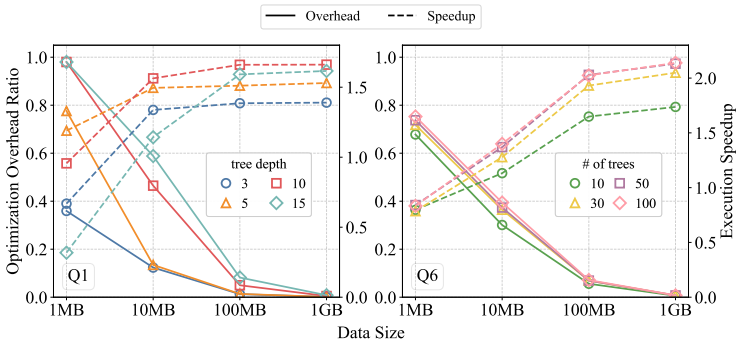


Fig. 16. Optimization Overhead Ratio vs. Execution Speedup

measure two key metrics: (1) *Optimization Overhead Ratio*, defined as the ratio of optimization overhead to the optimized execution time, and (2) *Execution Speedup*, defined as the ratio of the baseline execution time to the optimized execution time.

As depicted in Figure 16, the optimization overhead dominates at tiny scales (e.g., 1 MB), resulting in a performance reduction. However, applying optimization is practically superfluous in this scenario. For instance, Q1 with a depth-15 DT on 1 MB of data executes in less than 10 ms on a single thread. Since the latency is already within the interactive threshold, incurring an optimization is logically redundant. For small datasets (e.g., 10M), even when the optimization overhead ratio approaches 40%-60% for complex models, the reduction in execution time still outweighs the optimization overhead (reaching at least 1.15x). As the dataset size increases to 100MB and 1GB, the optimization overhead ratio decreases dramatically, representing a negligible fraction of the overall execution time.

In summary, our optimization targets scenarios where data scale makes execution latency a bottleneck. It proves highly effective for larger datasets ($>$ 100 MB) and remains practical for moderate sizes (10 MB), while for tiny datasets ($<$ 1 MB), the optimization is bypassed to rely on the efficient baseline.

8 Limitations and Future Work

ReTree introduces deviation in the accuracy of random forest regressor and binary classifier predicates. As shown in Figure 13, the deviation is minor in practice, yet it may still be a concern for applications with stringent accuracy requirements, where even slight errors could be significant. Additionally, ReTree currently does not support predicates with multi-class random forests or gradient-boosted decision trees (GBDT), limiting its applicability to a subset of tree-based models. Multi-class random forests require handling vector-valued probability distributions over multiple classes, and GBDTs construct trees sequentially, with each tree predicting the residuals of the cumulative model, making the trees non-equivalent. These structural differences prevent ReTree from being directly applied. We leave the extension of ReTree to these models as a promising direction for future work.

9 Related Work

In this section, we discuss prior work related to ReTree, including in-database prediction query execution, ML predicate optimization, and optimization of decision tree inference.

In-Database Prediction Query Execution. Supporting and optimizing prediction queries within databases is a promising research field at present. BigQuery [2] supports prediction queries using

TensorFlow [48] models. FiGO [4] builds model ensembles with PyTorch and selects appropriate models to balance query accuracy and performance. SmartLite [20] stores binary neural networks in database tables and converts model inference into SQL-based bit operations. InferDB [41] maps training data into an embedding space, storing the mapped data and predicted values in a table. Inference is then performed via an equi-`JOIN` with this table. LMFAO [43] and JoinBoost [12] convert ML algorithms into database aggregation operations. Given the maturity of Python's ML ecosystem, many databases support and optimize Python UDFs, including PG PL/Python [32], DuckDB [7], and MonetDB [37]. IMBridge [57, 58] further enhances the efficiency of Python UDFs by introducing one-off model context setup and desirable batching inference to optimize Python UDF execution. Nonetheless, these approaches do not perform internal optimizations on the model structure and are therefore orthogonal to ReTree. The two key optimizations of ReTree can be integrated into these systems.

ML predicate Optimization. Several systems aim to optimize queries involving ML predicates. CORE [54] and PP [21] construct lightweight proxy models or simplified predicates from complex models to rewrite queries and reduce computational overhead. EvaDB [52] avoids redundant UDF executions by caching UDF results across queries. Smart [10] leverages model metadata and statistics to employ ML predicates to prune irrelevant tuples. Raven [15, 31], on SQL Server [47], supports prediction queries using ONNX models. Raven builds a unified IR on top of ONNX by adding common relational algebra operators. This allows Raven to perform cross-optimizations such as predicate-based model pruning and model-projection pushdown. However, the optimization strategies of Raven and ReTree are different. Raven focuses on optimizing the input side of the model by using other simple predicates that share the same input columns with the ML predicate to prune unnecessary branches. In contrast, ReTree focuses on the output side of the model. Specifically, for decision tree or random forest predicates, ReTree eliminates redundant feature tests based on whether the inference result satisfies the predicate.

Optimization of Decision Tree Inference. Several techniques have been proposed to accelerate decision tree inference. Hummingbird [16, 24] transforms tree traversal into tensor operations, enabling the use of highly optimized tensor libraries. Other works, such as TreeBeard [33], Treelite [49], and lleaves [46] compile decision trees into efficient nested if-else structures. Additionally, systems like Tahoe [51], RAPIDS FIL [39, 40], and SilvanForge [34] aim to target decision tree models to run at peak performance on CPUs and GPUs. Despite their performance improvements, these systems primarily focus on accelerating inference for decision tree ensembles in general. They do not specifically target optimization opportunities arising from the integration of decision tree or random forest predicates within prediction queries.

10 Conclusion

In this work, we present ReTree, which eliminates redundant feature tests in ML predicates with decision trees and random forests. To eliminate the sibling-induced redundant feature tests, we propose the sibling-centric elimination with the merging-based subtree collapse method. To eliminate the ancestor-induced redundant feature tests, we propose the ancestor-centric elimination with the sliding-based subtree recombination method. We implement ReTree into DuckDB, and our experimental results show that ReTree accelerates the prediction query execution over DuckDB and outperforms alternative solutions like Raven and Smart.

Acknowledgments

This work was supported by the National Natural Science Foundation of China (No. 62272168), and the Natural Science Foundation of Shanghai (No. 23ZR1419900).

References

[1] Varun Babbar, Hayden McTavish, Cynthia Rudin, and Margo Seltzer. 2025. Near-Optimal Decision Trees in a SPLIT Second. In *Forty-second International Conference on Machine Learning (ICML)*.

[2] BigQuery. 2025. Make predictions with imported TensorFlow models. https://cloud.google.com/bigquery/docs/making-predictions-with-imported-tensorflow-models.

[3] Christoph Brücke, Philipp Härtling, Rodrigo D Escobar Palacios, Hamesh Patel, and Tilmann Rabl. 2023. TPCx-AI - An Industry Standard Benchmark for Artificial Intelligence and Machine Learning Systems. *Proceedings of the VLDB Endowment (PVLDB)* 16, 12 (2023), 3649–3661.

[4] Jiashen Cao, Karan Sarkar, Ramyad Hadidi, Joy Arulraj, and Hyesoon Kim. 2022. FiGO: Fine-Grained Query Optimization in Video Analytics. In *Proceedings of the 2022 International conference on management of data (SIGMOD)*. 559–572.

[5] Shaosheng Cao, XinXing Yang, Cen Chen, Jun Zhou, Xiaolong Li, and Yuan Qi. 2019. TitAnt: online real-time transaction fraud detection in Ant Financial. *Proceedings of the VLDB Endowment (PVLDB)* 12, 12 (2019), 2082–2093.

[6] Cornell University. 2025. Mathematical Programming I. https://people.orie.cornell.edu/dpw/orie6300/.

[7] DuckDB. 2025. From Waddle to Flying: Quickly expanding DuckDB's functionality with Scalar Python UDFs. https://duckdb.org/2023/07/07/python-udf.html.

[8] Léo Grinsztajn, Edouard Oyallon, and Gaël Varoquaux. 2022. Why do tree-based models still outperform deep learning on typical tabular data? *Advances in neural information processing systems (NeurIPS)* 35 (2022), 507–520.

[9] Hong Guan, Saif Masood, Mahidhar Dwarampudi, Venkatesh Gunda, Hong Min, Lei Yu, Soham Nag, and Jia Zou. 2023. A Comparison of End-to-End Decision Forest Inference Pipelines. In *Proceedings of the 2023 ACM Symposium on Cloud Computing (SoCC)*. 200–215.

[10] Yunyan Guo, Guoliang Li, Ruilin Hu, and Yong Wang. 2025. In-Database Query Optimization on SQL with ML Predicates. *The VLDB Journal* 34, 1 (2025), 12.

[11] Joseph M Hellerstein, Christoper Ré, Florian Schoppmann, Daisy Zhe Wang, Eugene Fratkin, Aleksander Gorajek, Kee Siong Ng, Caleb Welton, Xixuan Feng, Kun Li, et al. 2012. The MADlib Analytics Library: or MAD Skills, the SQL. *Proceedings of the VLDB Endowment (PVLDB)* 5, 12 (2012), 1700–1711.

[12] Zezhou Huang, Rathijit Sen, Jiaxiang Liu, and Eugene Wu. 2023. JoinBoost: Grow Trees over Normalized Data Using Only SQL. *Proceedings of the VLDB Endowment (PVLDB)* 16, 11 (2023), 3071–3084.

[13] Kaggle. 2025. State of Data Science and Machine Learning 2021. https://www.kaggle.com/kaggle-survey-2021.

[14] Kaggle. 2025. Walmart Recruiting - Store Sales Forecasting. https://www.kaggle.com/c/walmart-recruiting-store-sales-forecasting.

[15] Konstantinos Karanasos, Matteo Interlandi, Doris Xin, Fotis Psallidas, Rathijit Sen, Kwanghyun Park, Ivan Popivanov, Supun Nakandal, Subru Krishnan, Markus Weimer, Yuan Yu, Raghu Ramakrishnan, and Carlo Curino. 2020. Extending Relational Query Processing with ML Inference. In *Proceedings of the 10th Conference on Innovative Data Systems Research (CIDR)*.

[16] Dimitrios Koutsoukos, Supun Nakandala, Konstantinos Karanasos, Karla Saur, Gustavo Alonso, and Matteo Interlandi. 2021. Tensors: an abstraction for general data processing. *Proceedings of the VLDB Endowment (PVLDB)* 14, 10 (2021), 1797–1804.

[17] Arun Kumar, Jeffrey Naughton, Jignesh M. Patel, and Xiaojin Zhu. 2016. To Join or Not to Join? Thinking Twice about Joins before Feature Selection. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD)*. 19–34.

[18] Guoliang Li, Ji Sun, Lijie Xu, Shifu Li, Jiang Wang, and Wen Nie. 2024. GaussML: An End-to-End In-Database Machine Learning System. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. 5198–5210.

[19] Qiuru Lin, Sai Wu, Junbo Zhao, Jian Dai, Feifei Li, and Gang Chen. 2022. A Comparative Study of in-Database Inference Approaches. In *Proceedings of the 38th IEEE International Conference on Data Engineering (ICDE)*. 1794–1807.

[20] Qiuru Lin, Sai Wu, Junbo Zhao, Jian Dai, Meng Shi, Gang Chen, and Feifei Li. 2023. SmartLite: A DBMS-Based Serving System for DNN Inference in Resource-Constrained Environments. *Proceedings of the VLDB Endowment (PVLDB)* 17, 3 (2023), 278–291.

[21] Yao Lu, Aakanksha Chowdhery, Srikanth Kandula, and Surajit Chaudhuri. 2018. Accelerating Machine Learning Inference with Probabilistic Predicates. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD)*. 1493–1508.

[22] Christian Lülf, Denis Mayr Lima Martins, Marcos Antonio Vaz Salles, Yongluan Zhou, and Fabian Gieseke. 2023. Fast Search-by-Classification for Large-Scale Databases Using Index-Aware Decision Trees and Random Forests. *Proceedings of the VLDB Endowment (PVLDB)* 16, 11 (2023), 2845–2857.

[23] Zhenghua Lyu, Huan Hubert Zhang, Gang Xiong, Gang Guo, Haozhou Wang, Jinbao Chen, Asim Praveen, Yu Yang, Xiaoming Gao, Alexandra Wang, et al. 2021. Greenplum: a hybrid database for transactional and analytical workloads. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD)*. 2530–2542.

[24] Supun Nakandala, Karla Saur, Gyeong-In Yu, Konstantinos Karanasos, Carlo Curino, Markus Weimer, and Matteo Interlandi. 2020. A tensor compiler for unified machine learning prediction serving. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 899–917.
[25] Li GL Niu ZP. 2021. In-database AI Model Optimization. *Journal of Software* 32, 3 (2021), 622–635.
[26] ONNX. 2025. Introduction to ONNX. https://onnx.ai/onnx/intro.
[27] ONNX Runtime. 2025. ONNX Runtime. https://onnxruntime.ai/.
[28] Kweku-Muata Osei-Bryson. 2007. Post-pruning in decision tree induction using multiple performance measures. *Computers & operations research* 34, 11 (2007), 3331–3345.
[29] Matteo Paganelli, Paolo Sottovia, Kwanghyun Park, Matteo Interlandi, and Francesco Guerra. 2023. Pushing ML Predictions Into DBMSs. *IEEE Transactions on Knowledge and Data Engineering (TKDE)* 35, 10 (2023), 10295–10308.
[30] Qingfeng Pan, Jiahe Zhi, Chenyang Zhang, Chen Xu, Zhao Zhang, Anita Shao, Guanglei Bao, Qiu Cui, Xiaowei Chen, and Aoying Zhou. 2025. Machine Learning Inference Pipeline Execution using Pure SQL Based on Operator Fusion. In *2025 IEEE 41st International Conference on Data Engineering (ICDE)*. 3397–3410.
[31] Kwanghyun Park, Karla Saur, Dalitso Banda, Rathijit Sen, Matteo Interlandi, and Konstantinos Karanasos. 2022. End-to-End Optimization of Machine Learning Prediction Queries. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD)*. 587–601.
[32] PostgreSQL. 2025. PL/Python Functions. https://www.postgresql.org/docs/current/plpython-funcs.html.
[33] Ashwin Prasad, Sampath Rajendra, Kaushik Rajan, R Govindarajan, and Uday Bondhugula. 2022. Treebeard: An optimizing compiler for decision tree based ML inference. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 494–511.
[34] Ashwin Prasad, Sampath Rajendra, Kaushik Rajan, R Govindarajan, and Uday Bondhugula. 2024. SilvanForge: A Schedule-Guided Retargetable Compiler for Decision Tree Inference. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles (SOSP)*. 488–504.
[35] Zhen Qin, Le Yan, Honglei Zhuang, Yi Tay, Rama Kumar Pasumarthi, Xuanhui Wang, Michael Bendersky, and Marc Najork. 2021. Are neural rankers still outperformed by gradient boosted decision trees?. In *International Conference on Learning Representations (ICLR)*.
[36] J. Ross Quinlan. 1986. Induction of Decision Trees. *Machine learning* 1, 1 (1986), 81–106.
[37] Mark Raasveldt and Hannes Mühleisen. 2016. Vectorized UDFs in Column-Stores. In *Proceedings of the 28th International Conference on Scientific and Statistical Database Management (SSDBM)*.
[38] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an Embeddable Analytical Database. In *Proceedings of the 2019 international conference on management of data (SIGMOD)*. 1981–1984.
[39] RAPIDS AI. 2025. RAPIDS Forest Inference Library. https://github.com/rapidsai/cuml.
[40] RAPIDS AI. 2025. RAPIDS Forest Inference Library: Prediction at 100 million rows per second. https://medium.com/rapids-ai/rapids-forest-inference-library-prediction-at-100-million-rows-per-second-19558890bc35.
[41] Ricardo Salazar-Díaz, Boris Glavic, and Tilmann Rabl. 2024. InferDB: In-Database Machine Learning Inference Using Indexes. *Proceedings of the VLDB Endowment (PVLDB)* 17, 8 (2024), 1830–1842.
[42] Iqbal H Sarker. 2021. Machine Learning: Algorithms, Real-World Applications and Research Directions. *SN computer science* 2, 3 (2021), 160.
[43] Maximilian Schleich and Dan Olteanu. 2020. LMFAO: An engine for batches of group-by aggregates: layered multiple functional aggregate optimization. *Proceedings of the VLDB Endowment (PVLDB)* 13, 12 (2020), 2945–2948.
[44] scikit-learn. 2025. scikit-learn. https://scikit-learn.org/stable/.
[45] Ravid Shwartz-Ziv and Amitai Armon. 2022. Tabular data: Deep learning is not all you need. *Information Fusion* 81 (2022), 84–90.
[46] Simon Boehm. 2025. lleaves. hhttps://github.com/siboehm/lleaves.
[47] SQL Server. 2025. PREDICT (Transact-SQL). https://learn.microsoft.com/en-us/sql/t-sql/queries/predict-transact-sql?view=sql-server-ver15.
[48] TensorFlow. 2025. TensorFlow. https://www.tensorflow.org/.
[49] Treelite. 2025. model compiler for decision tree ensembles. https://treelite.readthedocs.io/en/latest/.
[50] Zhaomin Wu, Junhui Zhu, Qinbin Li, and Bingsheng He. 2023. DeltaBoost: Gradient Boosting Decision Trees with Efficient Machine Unlearning. *Proc. ACM Manag. Data (PACMMOD)* 1, 2, Article 168 (2023).
[51] Zhen Xie, Wenqian Dong, Jiawen Liu, Hang Liu, and Dong Li. 2021. Tahoe: tree structure-aware high performance inference engine for decision tree ensemble on GPU. In *Proceedings of the Sixteenth European Conference on Computer Systems (EuroSys)*. 426–440.
[52] Zhuangdi Xu, Gaurav Tarlok Kakkar, Joy Arulraj, and Umakishore Ramachandran. 2022. EVA: A Symbolic Approach to Accelerating Exploratory Video Analytics with Materialized Views. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD)*. 602–616.

[53] Jiani Yang, Sai Wu, Dongxiang Zhang, Jian Dai, Feifei Li, and Gang Chen. 2023. Rethinking Learned Cost Models: Why Start from Scratch? *Proc. ACM Manag. Data (PACMMOD)* 1, 4, Article 255 (2023).

[54] Zhihui Yang, Zuozhi Wang, Yicong Huang, Yao Lu, Chen Li, and X Sean Wang. 2022. Optimizing Machine Learning Inference Queries with Correlative Proxy Models. *Proceedings of the VLDB Endowment (PVLDB)* 15, 10 (2022), 2032–2044.

[55] Zhifeng Yang, Quanqing Xu, Shanyan Gao, Chuanhui Yang, Guoping Wang, Yuzhong Zhao, Fanyu Kong, Hao Liu, Wanhong Wang, and Jinliang Xiao. 2023. OceanBase Paetica: a hybrid shared-nothing/shared-everything database for supporting single machine and distributed cluster. *Proceedings of the VLDB Endowment (PVLDB)* 16, 12 (2023), 3728–3740.

[56] Zhenkun Yang, Chuanhui Yang, Fusheng Han, Mingqiang Zhuang, Bing Yang, Zhifeng Yang, Xiaojun Cheng, Yuzhong Zhao, Wenhui Shi, Huafeng Xi, et al. 2022. OceanBase: a 707 million tpmC distributed relational database system. *Proceedings of the VLDB Endowment (PVLDB)* 15, 12 (2022), 3385–3397.

[57] Chenyang Zhang, Junxiong Peng, Chen Xu, Quanqing Xu, and Chuanhui Yang. 2024. IMBridge: Impedance Mismatch Mitigation between Database Engine and Prediction Query Execution. In *Companion of the 2024 International Conference on Management of Data (SIGMOD)*. 456–459.

[58] Chenyang Zhang, Junxiong Peng, Chen Xu, Quanqing Xu, and Chuanhui Yang. 2025. Mitigating the Impedance Mismatch between Prediction Query Execution and Database Engine. *Proc. ACM Manag. Data (PACMMOD)* (2025).

Received October 2025; revised January 2026; accepted February 2026