JoinBoost: Grow Trees Over Normalized Data Using Only SQL

Zezhou Huang
zh2408@columbia.edu
Columbia University

Rathijit Sen
Rathijit.Sen@microsoft.com
Microsoft

Jiaxiang Liu
jl6235@columbia.edu
Columbia University

Eugene Wu
ewu@cs.columbia.edu
DSI, Columbia University

ABSTRACT

Although dominant for tabular data, ML libraries that train tree models over normalized databases (e.g., `LightGBM`, `XGBoost`) require the data to be denormalized as a single table, materialized, and exported. This process is not scalable, slow, and poses security risks. In-DB ML aims to train models within DBMSes to avoid data movement and provide data governance. Rather than modify a DBMS to support In-DB ML, is it possible to offer competitive tree training performance to specialized ML libraries...with only SQL?

We present `JoinBoost`, a Python library that rewrites tree training algorithms over normalized databases into pure SQL. It is portable to any DBMS, offers performance competitive with specialized ML libraries, and scales with the underlying DBMS capabilities. `JoinBoost` extends prior work from both algorithmic and systems perspectives. Algorithmically, we support factorized gradient boosting, by updating the Y variable to the residual in the *non-materialized join result*. Although this view update problem is generally ambiguous, we identify *addition-to-multiplication preserving*, the key property of variance semi-ring to support *rmse*, the most widely used criterion. System-wise, we identify residual updates as a performance bottleneck. Such overhead can be natively minimized on columnar DBMSes by creating a new column of residual values and adding it as a projection. We validate this with two implementations on `DuckDB`, with no or minimal modifications to its internals for portability. Our experiment shows that `JoinBoost` is 3× (1.1×) faster for random forests (gradient boosting) compared to `LightGBM`, and over an order of magnitude faster than state-of-the-art In-DB ML systems. Further, `JoinBoost` scales well beyond `LightGBM` in terms of the # features, DB size (TPC-DS SF=1000), and join graph complexity (galaxy schemas).

1 INTRODUCTION

Tree-based models—ranging from decision trees, random forests, and gradient boosting—are recursive space-partitioning algorithms for classification and regression [55]. They have shown exceptional effectiveness for tabular datasets, outperforming popular neural network models [37]. In fact, random forests and gradient boosting were rated the most popular model from 2019 to 2021 on Kaggle [11]. In response, the ML community has developed optimized tree-training libraries like `LightGBM` [48] and `XGBoost` [28], that offer user-friendly API, superior single-node performance, and compatibility with distributed frameworks like Dask [66] or Spark [77].

In practice, however, almost all tabular datasets are normalized and stored in a DBMS. Using ML libraries introduces performance, usability, and privacy drawbacks. First, the libraries expect a single (typically CSV) training dataset. Thus, a developer must denormalize the database into a "wide table" $R_{\bowtie}$, materialize and export $R_{\bowtie}$, and load it into the library. The join materialization cost is prohibitive for all but the simplest schemas and smallest databases—the 1.2GB IMDB dataset (Figure 3) is well over 1TB when fully

materialized due to N-to-N relationships. Second, managing the exported data as well as the separate execution frameworks adds considerable operational complexity and is error-prone [71, 73]. Third, exporting sensitive data can raise privacy concerns [2].

Ideally, we would "move computation to the data" and train tree-based models directly within the DBMS. This would let developers manage the entire data lifecycle—preparation, cleaning, analysis, and modeling—within a single DBMS, and benefit from the DBMSes' excellent security, performance, and scalability. To be broadly useful, we believe an initial In-DB solution should meet three criteria: (C1) be easily portable to any DBMS by translating ML algorithms into vanilla SQL queries, (C2) offer training performance comparable with the SOTA ML libraries (e.g., `LightGBM`), and (C3) scale to massive data warehouse sizes and complexity. Unfortunately, these criteria are often in tension. Common wisdom and prior results suggest that training tree models via SQL queries is portable but notoriously slow [38, 75, 77] as DBMSes do not employ model-specific optimizations. However, specialized optimizations [28, 48] or GPU acceleration [20, 39, 61] achieve competitive performance at the expense of portability to general DBMSes.

To optimize training over normalized schemas common in DBMSes, recent works in factorized ML [49, 50, 58, 68] avoid materializing joins by treating ML as semi-ring aggregation queries over the join graph; the aggregations can then be pushed through the joins. This approach potentially provides DBMSes with an edge over conventional ML libraries that face significant expenses for join materialization and export during training. Thus, it is intriguing to investigate the potential of training tree models in DBMSes using vanilla SQL, with factorized ML optimizations.

Nevertheless, the current applicability of factorized ML to tree-based models is restricted due to both algorithmic and system-related limitations. LMFAO [68] describes a factorized algorithm that is limited to decision trees, and does not support the widely used gradient boosting models. It rewrites the tree node split algorithm into a batch of group-by aggregations, and uses multi-query optimization to share work between the aggregations. Unfortunately, it does not support the residual updates needed for gradient boosting models. Even for single decision tree training, LMFAO fails to exploit considerable work-sharing between parent-child tree nodes. Finally, LMFAO lacks portability because its performance is tied to a compilation engine specially designed for its batch-wise query execution. In our experiments, despite using an off-the-shelf DBMS (DuckDB [64]), we train decision trees 1.9× faster than LMFAO.

To this end, we present `JoinBoost`, the first In-DB ML system designed for tree-based models (decision trees, random forests, and gradient boosting) that fulfill C1-3. Our main contribution lies in the system design and implementation: we study the feasibility of implementing `JoinBoost` as a Python library that translates ML algorithms into simple Select-Project-Join-Aggregation (SPJA) queries and employs factorized ML as pure query rewriting.

Such design ensures portability (C1) and scalability (C3), as the translated SQL queries can run on any single node or cloud-based DBMS. Nonetheless, this design poses significant challenges. Algorithmically, previous factorized ML systems do not support gradient boosting. Furthermore, while earlier factorized ML systems have demonstrated good performance, their success is closely linked to specialized execution engines. It's unclear whether implementing factorized ML as pure query rewriting can enable existing DBMSes to deliver performance (C2) competitive to specialized ML libraries.

To support factorized gradient boosting, each iteration trains a decision tree on the *residuals* of the prior tree, requiring updates to Y in the denormalized relation $R_{\bowtie}$; however, $R_{\bowtie}$ *is not materialized in factorized ML*, and the view update generally requires $O(R_{\bowtie})$ [5]. To address this, for snowflake schema, we exploit 1-1 mapping between $R_{\bowtie}$ with the fact table F, to update Y in F directly. For the more complex galaxy schema, we found that despite challenges in directly updating Y in $R_{\bowtie}$, the tree training only relies on aggregates of Y (e.g., count, sum) that could be updated efficiently. Our technique identifies *addition-to-multiplication preserving*, the key property to efficiently update aggregates, by rewriting residual updates over Y in $R_{\bowtie}$ into a join with an *update relation U*. However, each boosting iteration generates a new *update relation* that may create cycles and doesn't scale with the number of iterations. To address this, we introduce *Clustered Predicate Tree* (CPT) that restricts the tree splits to attributes within the same cluster. This enables us to train gradient boosting on IMDB datasets, which was previously prohibitive due to the large size of $R_{\bowtie}$ ($>1TB$).

To assess the viability of implementing factorized ML with pure SQL, we conduct extensive experiments on DuckDB and a popular commercial DBMS. Although columnar DBMS has the potential to compete with specialized ML libraries, we find residual updates as the major bottleneck for gradient boosting in current columnar DBMSes: Residual updates require sequential writes to all values in a column. However, this process is not efficient in existing DBMSes due to a combination of columnar compression, write-ahead-log (WAL), and concurrency control (CC); these are fundamental to DBMSes but unnecessary for gradient boosting. Disabling WAL, CC, and compression in existing DBMSes directly is challenging as they are deeply integrated into the codes. To match the performance of LightGBM, update performance must be similar to a parallelized write to an in-memory byte array.

To minimize DBMS overheads for residual updates (C2) without compromising portability (C1), we explore the approach to create a new column of residual values and adding it as a projection [74] on columnar DBMSes. We emulate this on DuckDB in two ways, with no or little modification to its internals. First, by leveraging DuckDB Pandas API [12], we store the fact table in a Pandas dataframe, use DuckDB for joins and aggregations, and update the residual by creating a new Pandas dataframe column. This results in a ~15× improvement in updates, making gradient boosting competitive without modifying DuckDB. However, one drawback is a ~1.6× slowdown in aggregations due to the DuckDB-Pandas interop overhead. To explore the full potential, we further modify DuckDB internals to support column swaps between DuckDB tables. We update residuals by swapping the existing residual column with the newly created one, making JoinBoost 1.1× faster than LightGBM. Lastly, we simulate column swap on a commercial DBMS, and see a potential 15×

improvement. Such modifications, being both feasible (< 100 LOC on DuckDB) and effective, provide direction for other closed-source columnar DBMSes to support in-DB gradient boosting efficiently.

Finally, we apply optimizations to further improve JoinBoost performance. Algorithmically, we enhance prior batch optimization [68] by sharing intermediate results (materialized as tables in DBMSes) across batches (tree nodes), leading to a 3× improvement. System-wise, we leverage inter-query parallelism among SQL queries, accelerating the gradient boosting training by 28% and random forest by 35%. We conduct extensive experiments with JoinBoost on various DBMS backends (local and cloud), using datasets with varying feature numbers, sizes (TPC-DS with SF 10→1000), and join graph complexity (galaxy schemas), against SOTA ML libraries (LightGBM,XGBoost,Sklearn) and In-DB systems (LMFAO,MADLib). **On a single node, JoinBoost trains a gradient boosting model with 100 trees** 1.1× **faster than LightGBM, and is** 3× **faster for random forests. On multiple nodes, JoinBoost outperforms the Dask [66] versions of LightGBM and XGBoost by** >9×**. JoinBoost easily scales in the # of features, join graph complexity, and dataset size (TPC-DS SF=1000 in Section 6.2), whereas LightGBM encounters memory limitations.**
Note: The paper is self-contained and references to appendices can be skipped, or can be found in the technical report [13].

2 RELATED WORK

Tree-based ML Libraries. Random forest [23] and gradient boosting [34] are the de-facto ensemble models supported by almost all standard ML libraries including Sklearn [63], TensorFlow [16] and Keras [30]. The leading Tree-based ML libraries are LightGBM [48] and XGBoost [28], both of which are highly optimized and outperform others as per previous benchmarks [19]. According to Kaggle 2021 surveys [11], among all the commonly used ML libraries, XGBoost and LightGBM are ranked 4th and 6th respectively for popularity (while all the top 3 also support Tree-based ML). However, none of them apply factorized ML for normalized datasets.

In-DB ML systems. Most in-DB ML works [33, 38, 75] focus on extending DBMSes (e.g., PostgreSQL for MADLib) to support linear algebra using UDFs and user-defined types. However, these are not needed for tree-based models that only rely on simple aggregations.

Other work optimizes ML training by leveraging specialized features (distributed execution [22, 44, 45, 51, 77] or GPU acceleration [20, 39, 53, 61]) of a specific DBMS. Although we focus on optimizing the single-node CPU setting via SQL rewrites, JoinBoost can run on any DBMS and benefit from its optimizer and executor. For instance, Section 6.2 scales decision tree training on TPC-DS SF=1000 using a cloud DBMS. We leave further DBMS-, data-, and model-specific optimizations to future work.

In practice, cloud vendors (Azure [1], Redshift [4], BigQuery [54], Snowflake [3]) offer SQL syntax to train tree models. Under the covers, however, they still materialize and export join results to an external library (LightGBM or XGBoost). Thus, we do not consider them in-DB ML from a performance or portability standpoint.

Factorized ML systems. Factorized ML is optimized to train models over normalized databases. There are two classes. The first class translates ML models as aggregations over an appropriately designed semi-ring, and pushes aggregations through joins to achieve asymptotically lower time complexity. They support many popular

models (ridge regression [68], SVM [49], factorization machine [68]), and approximate others (k-means [32], GLM [43]). Of these, only `LMFAO` [68] supports decision trees, albeit in a limited way (see Section 6.4). Further, most factorized ML works build specialized query optimizers and executors from scratch, which hinders portability (of the systems and performance wins) to existing DBMSes. `JoinBoost` extends factorized ML to tree models via vanilla SQL queries, and shows competitive performance with `LightGBM`.

The second class factorizes linear algebra operations over joins [27, 52, 78] by carefully iterating over the tables, caching intermediates, and combining their results. Compared to the first class, these works support any model that can be formulated as linear algebra statements, such as K-means and GLM (without approximation). However, they only reduce space and not time complexity, and are worse than naive denormalization when data fits in memory [52].

3 BACKGROUND

We provide the background of factorized tree-based models.

3.1 Annotated Relations and Message Passing

This section provides an overview of annotated relations and message passing fundamental to factorized query execution [18, 59].

Data Model. We use the traditional relational data model with the following notations: Given relation R, let uppercase A be an attribute, $dom(A)$ be its domain, $S_R = [A_1, \cdots, A_n]$ be its schema, $t \in R$ as a tuple of R, and $t[A]$ be tuple t's value of attribute A. For clarity, we include the schema in the square bracket followed by the relation $R[A_1, \cdots, A_n]$. The domain of R is the Cartesian product of attribute domains $dom(R) = dom(A_1) \times \cdots \times dom(A_n)$.

Annotated Relations. The annotated relational model [36, 46, 57] maps each $t \in R$ to a commutative semi-ring $(D, \oplus, \otimes, 0, 1)$, where D is a set, $\oplus$ and $\otimes$ are commutative binary operators closed over D, and $0/1$ are the zero/unit elements. These annotations form the basis of provenance semi-rings [36], and are amenable query optimizations based on algebraic manipulation. Different semi-ring definitions support different aggregation functions, ranging from standard statistical functions to ML models. For instance, the natural numbers semi-ring $(\mathbb{N}, +, \times, 0, 1)$ allows for integer addition and multiplication, and supports the *COUNT* aggregate. For an annotated relation R, let $R(t)$ denote tuple t's annotation.

Semi-ring Aggregation Query. Semi-ring aggregation queries can now be re-expressed over annotated relations by translating group-by (general projection) and join operations respectively into $+$ and $\times$ operations over the semi-ring annotations:

$$(\gamma_{\mathbf{A}} R)(t) = \sum \{R(t_1) \mid t_1 \in R, t = \pi_{\mathbf{A}}(t_1)\} \tag{1}$$

$$(R \bowtie T)(t) = R(\pi_{S_R}(t)) \otimes T(\pi_{S_T}(t)) \tag{2}$$

(1) Each group-by result annotation in $\gamma_{\mathbf{A}} R$ sums all the annotations in its input group. (2) In $R \bowtie T$, the annotation of each join result is the product of annotations from corresponding tuples in R and T.

Aggregation Pushdown. The key optimization in factorized query execution [18, 69] is to distribute aggregations (additions) through joins (multiplications). Consider $\gamma_D(R[A, B] \bowtie S[B, C] \bowtie T[C, D])$. The naive execution first materializes the join and then computes the aggregation. This costs $O(n^3)$ where n is each relation's cardinality. An alternative evaluation could apply aggregation (addition) to R before joining (multiplication) with S, aggregate again before

joining with T, and then apply a final aggregation. The largest intermediate result, and thus the join complexity, is now $O(n)$.

$$\gamma_D(\gamma_C((\gamma_B R[A, B]) \bowtie S[B, C]) \bowtie T[C, D])$$

Message Passing. Given a join graph, the above optimization can be viewed as *Message Passing* [62]. While *Message Passing* supports general SPJA queries [42], it is sufficient for tree models to restrict ourselves to SPJA queries with zero (γ) or one (γ_A) group-by attribute. Message passing operates over a tree that spans the join graph. For the root, we can pick any relation (taking cost models into account) that contains the grouping attribute. Then we direct all the edges of the join graph toward the root to form a tree[1].

Starting from the leaf, we send messages along its path to the root. Each message is computed as: (1) Join the relation with all incoming messages from children relations, if any. This blocks until all children have emitted messages. Then (2) let $\mathbb{A}$ be the attributes in the current relation that are also contained in the remaining relations in the path to the root. Compute $\gamma_{\mathbb{A}}$ over the previous join result, and emit the result to the parent relation.

Consider again $\gamma_D(R[A, B] \bowtie S[B, C] \bowtie T[C, D])$ along the join graph: $R - S - T$. If we choose T as the root, then the directed join graph is $R \rightarrow S \rightarrow T$ and the messages are:

$$m_{R \rightarrow S} = \gamma_B R[A, B], \qquad m_{S \rightarrow T} = \gamma_C(m_{R \rightarrow S} \bowtie S[B, C])$$

Once the root receives and joins with all messages, it performs **absorption**, which simply applies the final group-by: $\gamma_D(m_{S \rightarrow T} \bowtie T[C, D])$. In some cases, the aggregate result is already part of the semi-ring (e.g., *COUNT* and the natural numbers semi-ring); in other cases, such as tree-based models, the semi-ring decomposes the training metric into their constituent statistics, so we combine them in the final annotation to restore the metric (see next section).

3.2 Tree-based Models

In this section, we describe the traditional tree-based models. The algorithms are based on CART [24], and its extensions (e.g., bagging and boosting) follow standard ML textbooks [55].

Decision Tree. Decision tree maps (predicts) target variable Y from a set of features $\mathbf{X}$. Internally, it maintains a tree structure of selection predicates, with each edge containing a predicate σ and selecting data based on the conjunction of predicates along the path from the root. Leaf nodes associate with a prediction value $p \in dom(Y)$. The selection predicates by leaves are mutually exclusive and collectively exhaustive in $Dom(\mathbf{X})$. For $t \in Dom(\mathbf{X})$, decision tree predicts by traversing the edges where t meets the edge predicate until a leaf is reached, then outputs its prediction p.

Training a Decision Tree over relation R with features $\mathbf{X}$ and target variable Y where $\mathbf{X} \cup Y \subseteq S_R$ involves recursively splitting R to minimize a criterion $c(\cdot)$. For regression, the popular criterion is Variance for root mean square error (*rmse*), while Gini Impurity, Entropy, and Chi-Square are common for classification. Numerical attribute splits use inequality ($\sigma_{A>v}, \bar{\sigma}_{A \leq v}$), and categorical attribute splits use equality ($\sigma_{A=v}, \bar{\sigma}_{A \neq v}$) or set-based ($\sigma_{A \in V}, \bar{\sigma}_{A \notin V}$) predicates. Given the criteria $c(\cdot)$ and split σ, the reduction of criteria after the split is $c(R) - (c(\sigma(R)) + c(\bar{\sigma}(R)))$. Tree growth could be depth-wise or best-first [72]. Depth-wise growth splits the tree node with the least depth, and best-first growth splits the tree node

---

[1]The algorithm applies to acyclic join graphs. Cyclic join graphs can be pre-joined into acyclic join graphs using standard hypertree decomposition [18, 47].

Semi-ring	Zero/One	Operator	Lift
Variance	**0:** $(0, 0, 0)$	$(c_1, s_1, q_1) \oplus (c_2, s_2, q_2) = (c_1 + c_2, s_1 + s_2, q_1 + q_2)$	
$(\mathbb{Z}, \mathbb{R}, \mathbb{R})$	**1:** $(1, 0, 0)$	$(c_1, s_1, q_1) \otimes (c_2, s_2, q_2) = (c_1 c_2, s_1 c_2 + s_2 c_1, q_1 c_2 + q_2 c_1 + 2 s_1 s_2)$	$(1, y, y^2)$
Class Count	**0:** $(0, ..., 0)$	$(c_1, c_1^1, ..., c_1^k) \oplus (c_2, c_2^1, ..., c_2^k) = (c_1 + c_2, c_1^1 + c_2^1, ..., c_1^k + c_2^k)$	
$(\mathbb{Z}, \mathbb{Z}, ..., \mathbb{Z})$	**1:** $(1, ..., 0)$	$(c_1, c_1^1, ..., c_1^k) \otimes (c_2, c_2^1, ..., c_2^k) = (c_1 c_2, c_1^1 c_2 + c_1 c_2^1, ..., c_1^k c_2 + c_1 c_2^k)$	$(1, 0, ..., c^y = 1, ..., 0)$

Table 1: Example commutative semi-rings for decision trees. Variance semi-ring supports regression criteria like Reduction in Variance; Class Count semi-ring supports classification criteria like Gini Impurity, Information Gain, and Chi-Square.

greedily with the largest criteria reduction. Finally, each leaf prediction is the average Y for regression or mode Y for classification.

Algorithm 1 presents the training algorithm. Given relation $R_\bowtie$ (with Y encoded in annotations), features $\mathbf{X}$, and the maximum number of leaves, the algorithm iteratively calls GetBestSplit to find the best next split (L3,8,9). GetBestSplit iterates through all features, evaluates its best split (based on *reduction of criteria*), and returns the best split across all features (L11-16). For best-first growth, a priority queue (L2) sorts leaf nodes descendingly based on *reduction of criteria*. In each iteration, the algorithm splits the best leaf node (L7), creating two tree nodes that partition the parent node. It then finds the best split for each new node and pushes them to the queue (L8,9). Training ends when the number of leaves reaches the maximum (L5). Leaf node predictions are computed, and the tree is returned (L10). Note that Algorithm 1 is a minimal decision tree for clarity, and practical decision trees have additional parameters like regularization (Appendix B). During training, evaluating the best split (L14) is the most computationally expensive.

Algorithm 1: Decision tree training algorithm. L14 (underlined) is the most computationally expensive.

---

1 **Function** DecisionTree($R_\bowtie$, $\mathbf{X}$, *maxLeaves*)
2   *numLeaves*, *pq* $\leftarrow$ 1, *priorityQueue*();
3     $c, \sigma, X, R_\bowtie, root \leftarrow$ GetBestSplit($R_\bowtie$, $\mathbf{X}$, *null*, *null*);
4   *pq.push*($c, \sigma, X, R_\bowtie, root$);
5   **while** *numLeaves*++ < *maxLeaves* **do**
6   // get the best split with the highest c among leaves;
7        $\_, \sigma_{parent}, X, R, parent \leftarrow pq.pop$();
8   *pq.push*(GetBestSplit(R, $\mathbf{X}$, σ_{parent}, *parent*));
9   *pq.push*(GetBestSplit(R, $\mathbf{X}$, $\neg\sigma_{parent}$, *parent*));
10   **return** *addPrediction*($R_\bowtie$, *root*)
11 **Function** GetBestSplit(R, $\mathbf{X}$, σ, *parent*)
12     $c^*, \sigma^*, X^*, node \leftarrow 0, null, null, createNode(\sigma, parent)$;
13   **for** *each feature* $X \in \mathbf{X}$ **do**
14   <u>$\sigma, c \leftarrow$ best split and criteria reduction for X over $\sigma(R)$;</u>
15   **if** $c > c^*$ **then** $c^*, \sigma^*, X^* \leftarrow c, \sigma, X$;
16   **return** $c^*, \sigma^*, X^*, \sigma(R), node$

Bagging and Boosting. Big, deep decision trees risk overfitting. Ensemble methods combine multiple smaller or shallower decision trees to create a more robust model. Popular tree-based ensemble models include random forests [23] (bagging) and gradient boosting [34] (boosting). Random forests parallelly train decision trees on samples of R and features and aggregate their predictions (e.g., average) for the final result. Gradient boosting sequentially trains decision trees based on preceding trees' residuals.

3.3 Factorized Decision Tree

We now introduce factorized decision trees over joins. The training process follows Algorithm 1, but we optimize the computation of criteria reduction (L14), the most computationally intensive part, by avoiding join materialization. We consider a database with a set of relations $\mathcal{R} = \{R_1, R_2, ..., R_n\}$, and aim to train a decision tree over $R_\bowtie = R_1 \bowtie R_2 ... \bowtie R_n$ with a set of features $\mathbf{X}$ and target variable Y. Let R_Y be the relation that contains Y (or pick one if Y is in many relations as a join key). For simplicity, we will assume natural join, set semantics, and the Variance split criterion; Appendix B describes extensions to theta and outer joins, bag semantics, and formulas for classification semi-rings. Factorized learning avoids materializing $R_\bowtie$ by expressing the criterion as semi-ring aggregation queries, and applying aggregation pushdown over join. We first describe semi-ring annotations for trees and then the computation of criterion.

Tree Semi-rings. We now illustrate how to use variance semi-ring (Table 1) to compute the *Variance* for regression. Each semi-ring defines a $lift(\cdot)$ function [57] that annotates a base tuple with its appropriate semi-ring element. Variance semi-ring lifts $t \in R_Y$ with $(1, t[Y], t[Y]^2)$, and t from the remaining relations with the 1 element $(1, 0, 0)$. During message passing, the annotations are combined via $\oplus$ and $\otimes$ as defined in Table 1. The aggregated semi-ring $\gamma(R_\bowtie)$ then forms a 3-tuple (C, S, Q) that denotes the count, sum of Y, and the sum of squares of Y. The variance statistic can be derived from this aggregated semi-ring as $variance = Q - S^2/C$. Thus, any filter aggregation query over the join graph can be expressed using message passing and lightweight post-processing.

EXAMPLE 1. *Consider $\mathcal{R} = \{R, S, T\}$ in Figure 1a with variance semi-ring, join graph $R - S - T$ and target variable $Y = B$. To compute the variance over $R_\bowtie$, the naive solution is to materialize $R_\bowtie$ (Figure 1b), then compute the variance (=4) over B. Instead, we compute $(C, S, Q) = \gamma(R \bowtie S \bowtie T) = (8, 16, 36)$ and $variance = Q - S^2/C = 4$.*

To find the best split, we similarly use variance semi-ring compute the *reduction in variance*. Given a tree node with total count (C_{total}) and sum of Y (S_{total}) over R as precomputed constants, a split $\sigma{:}f = v$ for a categorical feature f and a value v yields a variance reduction calculated as $-S_{total}^2/C_{total} + S_\sigma^2/C_\sigma + (S_{total} - S_\sigma)^2/(C_{total} - C_\sigma)$. Here, C_σ and S_σ denote the count and sum of Y over $\sigma(R)$, respectively. The optimal split maximizes this variance reduction, evaluated per feature. For numerical features, we further require prefix sums of C, S (ordered by feature value) to evaluate predicates with inequality (e.g., $\sigma = f < v$); SQL window functions can achieve this. We provide an example SQL to find the best split, and the full derivation is in Appendix A due to space limit.

EXAMPLE 2. *Given relation R and numerical feature A. To find its best split point v (i.e., $\sigma{:}A < v, \neg\sigma{:}A \geq v$), we use the following SQL:*
```
SELECT A, -({$stotal}/{$ctotal}) * {$stotal} + (s/c) * s
```

(a) Relations annotated with variance semi-ring, join graph $R - S - T$, target variable B and features C,D.

(b) Naive join $R_{\bowtie}$ and aggregation query $\gamma(R_{\bowtie})$.

(c) Message passing for aggregation query $\gamma_C(R_{\bowtie})$. The root node is dotted in green.
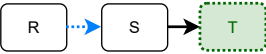
(d) Message sharing between aggregation queries $\gamma_D(R_{\bowtie})$ and $\gamma_C(R_{\bowtie})$. The reusable message is dotted in blue.

Figure 1: Factorized decision stump training.

```
     + ({$stotal} - s)/({$ctotal} - c) * ({$stotal} - s) AS criteria
FROM (SELECT A, SUM(c) OVER(ORDER BY A) as c, SUM(s) OVER(ORDER BY A) as s
     FROM (SELECT A, sum(Y) as s, COUNT(*) as c FROM R GROUP BY A))
ORDER BY criteria DESC LIMIT 1;
```

Message Caching. Training a decision stump requires computing the set of aggregation queries grouped by each feature: $\{\gamma_X(R_{\bowtie})|X \in \mathbf{X}\}$. Executing each independently via message passing is wasteful due to reusable messages. Consider the example in Figure 1:

EXAMPLE 3. *Let $\mathbf{X} = \{C, D\}$, and suppose we first aggregate on C (Figure 1c). We choose S as the message passing root because it contains C; we then pass messages m_1 from R to S and m_2 from T to S, and absorb the messages into S. We then aggregate on D (Figure 1d). We choose T as the message passing root, pass m_1 from R to S, m_3 from S to T, and absorb into T. The two queries can reuse m_1.*

Recent work [42] shows that a simple caching scheme that materializes all messages between relations in the join graph (in both directions) is effective in various analytical workloads. The factorized learning system LMFAO [68] also optimizes batch queries for splitting a single decision tree node. For decision trees, where each query groups by ≤1 feature, its optimizations are equivalent to this simple caching scheme. However, LMFAO optimizes a single decision tree node's batch of queries, missing work-sharing across tree nodes and not supporting residual updates for boosting.

4 FACTORIZED GRADIENT BOOSTING

This section delves into the algorithms for factorized gradient boosting, which iteratively trains the next tree to predict the residuals from preceding trees. Recall from Section 3.2 that, each leaf l in the decision tree is associated with a predicate $l.\sigma$ and prediction $l.p$. We aim to update, for each leaf l, the target variable of $t \in l.\sigma(R_{\bowtie})$ to the residual $t[\mathcal{E}] = t[Y] - l.p$. *The fundamental challenge is to train decision tree over the updated Y in $R_{\bowtie}$ without materializing $R_{\bowtie}$.*

One tempting approach is to treat this as a view update problem: we update Y in the $R_{\bowtie}$ view, translate it into updates over base relations, re-lift the updated base relations, and train the next factorized decision tree. However, view updates are susceptible to "side-effects" [5]: for $t \in R_Y$, $y = t[Y]$ can be duplicated in $R_{\bowtie}$ due to 1-N (or M-N) joins. If one duplicate is updated to a new value y', updating $t[Y] = y'$ in R_Y would inadvertently cause other duplicates to be updated as well (as "side-effects"). To address it, new tuples are created in base relations for new mappings. These "side-effects" are particularly problematic for "residual updates", as the full Y column in $R_{\bowtie}$ is updated, requiring $O(|R_{\bowtie}|)$ new tuples.

We address these issues for snowflake (a single fact table) and galaxy schemas (multiple fact tables)—arguably the most common database schema forms. Snowflake schemas exhibit a 1-to-1 relationship between the fact table F and $R_{\bowtie}$, and we exploit this to efficiently updates Y over F. We then discuss the system challenges when performing full-column updates. These techniques do not apply to galaxy schemas due to the M-N relationships between the fact tables. While directly updating Y in $R_{\bowtie}$ remains challenging, we identify the *addition-over-multiplication preserving* property that allows us to directly update semi-ring aggregates derived from updated Y, and use this to efficiently support the popular *rmse* criteria.

4.1 Snowflake Schemas

Snowflake schemas have one fact table F and N-to-1 paths along the dimension tables; this means that F is 1-1 with $R_{\bowtie}$[2] and we can directly update F. Let us first assume that Y is in F (i.e., $R_Y = F$). The main challenge is to translate a leaf l's predicate $l.\sigma$, which may refer to dimension attributes, into one over F. We do so by translating $l.\sigma$ as semi-join predicates[3] over F. Given join path $F - D_1 - ... - D_k$, and $\sigma(D_i)$, we "move" the predicate to be $\sigma'(D_{i-1})$:

$$D_{i-1} \bowtie \sigma(D_i) = (D_{i-1} \ltimes \sigma(D_i)) \bowtie D_i = \sigma'(D_{i-1}) \bowtie D_i$$

where σ' is over the join keys $\mathcal{J} = S_{D_{i-1}} \cap S_{D_i}$. Note that σ doesn't have to be equality-based but can be of any arbitrary type: σ is applied to $\pi_{\mathcal{J}}(\sigma(D_i))$ to identify the matching join keys for the semi-join. If Y is in a dimension table D (i.e., $R_Y \neq F$), we join the relations along the path from F to D and project all attributes in F along with Y. This reduces to the first case above.

System Challenge: Since $|F| = |R_{\bowtie}|$, factorized gradient boosting *does not offer an asymptotic advantage over specialized ML libraries.* Theoretically, if the performance is comparable, it could still benefit from the parallelization, scaling, and administrative features of DBMSes. However, in practice, we find that bulk updates to $F.Y$ are a major bottleneck (Section 5.3.2) for existing DBMSes. The next

---

[2]We assume no missing join keys, or use left outer join to maintain 1-1 relationship.
[3]We treat left semi-joins as filters over the left relation so its annotations don't change.

section dives into these system challenges, and explores system design, logical, and physical optimizations to address them.

4.2 Galaxy Schemas

Galaxy schemas model M-N relationships between multiple fact tables. They are prevalent in enterprise settings [67]; the recent "semantic layers" trend [9, 15, 26, 41] (pre-defined denormalized views) have made analysis and ML over galaxy schemas more accessible. However, galaxy schemas induce the 1-N relationship between R_Y and $R_{\bowtie}$ that causes side-effects [5] during residual updates.

We observe that individual Y values are not needed for training—the split criteria only refers to *semi-ring aggregates* over the updated Y values (Section 3.3), and can potentially be updated efficiently. For instance, for a leaf l with original aggregates $c = \sum 1, s = \sum y$ over $l.\sigma(R_{\bowtie})$, we can directly update the sum of residuals as $\sum(y-l.p) = \sum y - \sum l.p = s - c \times l.p$, without referencing individual Y values.

Unfortunately, this approach is not efficient for arbitrary split criteria. To review, factorized ML maps (via $lift(\cdot)$) y in the real number semi-ring $\mathbb{R}(\mathcal{R}, +, \times, 0, 1)$ (for base relation) to another semi-ring $\mathbb{S}(\mathcal{S}, \oplus, \otimes, 0', 1')$ (e.g., variance semi-ring for *rmse*), and computes $\sum lift(y)$ over $R_{\bowtie}$ with aggregation pushdown. To directly update aggregates given leaf l, we want to find a function $f: (\mathcal{S}, \mathcal{R}) \to \mathcal{S}$, such that f takes the original aggregates $\sum lift(y)$ (without referencing individual y) and additive inverse of prediction $l.p$[4] as input, and outputs updated aggregates: $f(\sum lift(y), -l.p) = \sum lift(y - l.p)$. To maintain our asymptotic benefits, both f and semi-ring operation/annotation should be of constant time/size.

Such f does not exist for all semi-rings. For example, *mean absolute error* (*mae*) relies on the count and sum of signs: $\sum 1, \sum sign(y)$. The naive semi-ring would track $\sum lift(y) = (\sum 1, \sum sign(y))$. However, f doesn't exist because given the leaf l, $\sum sign(y - l.p)$ cannot be solely decided by $\sum 1, \sum sign(y), -l.p$, as it depends on individual $y - l.p$. A naive solution uses a semi-ring that collects all y values, but its annotation size will be $O(|R_{\bowtie}|)$ and defeats the purpose of factorization. To this end, we identify a sufficient property for constant sized semi-ring $\mathbb{S}$ to construct f:

DEFINITION 1. *Addition-to-Multiplication Preserving. Given two semi-rings $\mathbb{R}(\mathcal{R}, +, \times, 0, 1)$, $\mathbb{S}(\mathcal{S}, \oplus, \otimes, 0', 1')$, a lift function $lift(\cdot) : \mathcal{R} \to \mathcal{S}$ is addition-to-multiplication preserving from $\mathbb{R}$ to $\mathbb{S}$ if $lift(d_1 + d_2) = lift(d_1) \otimes lift(d_2)$ for any $d_1, d_2 \in D$.*

Based on the property, we can construct f as:

$$f\left(\sum lift(y), -l.p\right) = \left(\sum lift(y)\right) \otimes lift(-l.p) \quad (f \text{ definition})$$
$$= \sum \left(lift(y) \otimes lift(-l.p)\right) \quad (\otimes \text{ distributive})$$
$$= \sum lift(y - l.p) \quad (\text{Add-to-Mul Preserving})$$

It is easy to verify that, the variance semi-ring (and lift) in Table 1 satisfies the property: $lift(y - l.p) = (1, y - l.p, l.p^2 + y^2 - 2l.p \times y) = (1, y, y^2) \otimes (1, -l.p, l.p^2) = lift(y) \otimes lift(-l.p)$. In contrast, there is no known $lift$ and constant-sized semi-ring with this property for *mae*. Section 7 describes a possible future direction using polynomial approximations [43] of the semi-ring aggregates. To keep the text concrete, the subsequent text will refer to the *rmse* criteria.

4.2.1 Update Relation:
For decision tree and each of its leaf l, we apply the corresponding f that multiplies aggregates over $l.\sigma(R_{\bowtie})$ with $lift(-l.p)$. We model this as an *Update Relation* U joins with
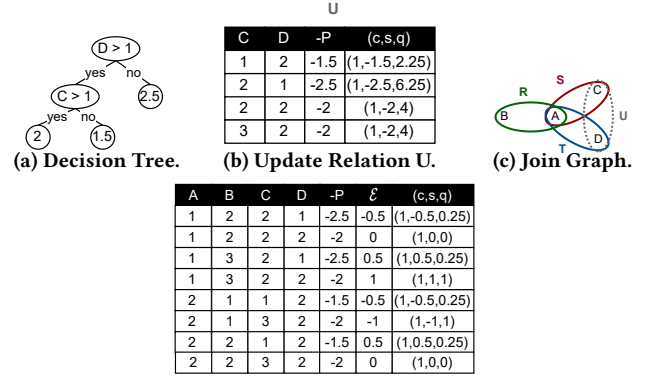

(a) Decision Tree.   (b) Update Relation U.   (c) Join Graph.

(d) Materialized $R_{\bowtie}$ with prediction additive inverse $-P$, residual $\mathcal{E} = Y - P$ and annotation $(1, \epsilon, \epsilon^2)$ lifted on $\mathcal{E}$ for the next tree.

Figure 2: Update Relation U for decision tree residual updates over the non-materialized $R_{\bowtie}$. Each tuple in $U \bowtie R_{\bowtie}$ has the same annotation as the materialized $R_{\bowtie}$ lifted on $\mathcal{E}$. Therefore, the materialization of $R_{\bowtie}$ can be avoided.

$R_{\bowtie}$. U is constructed as follows: let $\mathbf{A}$ be the set of attributes referenced by $l.\sigma$ for any leaf l, and U be the projection $\pi_{\mathbf{A}}(R_{\bowtie})$, along with a column $-P$ of the additive inverse of leaf prediction (unique as leaf predicates are non-overlapping). Naively, the next boosting lifts the residual using $lift_{Y-P}(R_{\bowtie} \bowtie U)$[5], where $R_{\bowtie} \bowtie U$ has to be materialized. Instead, we rewrite this into $lift_Y(R_{\bowtie}) \bowtie lift_{-P}(U)$:

PROPOSITION 4.1. *Let $t \in U \bowtie R_{\bowtie}$. $lift_{\mathcal{E}}(U \bowtie R_{\bowtie})(t) = (lift_Y(R_{\bowtie}) \bowtie lift_{-P}(U))(t)$ if $lift(\cdot)$ is addition-to-multiplication preserving.*

PROOF. Consider any $t \in U \bowtie R_{\bowtie}$ with $y = t[Y]$ and $-p = t[-P]$. Naively, we materialize $\mathcal{E} = Y - P$, then lift $lift_{\mathcal{E}}(U \bowtie R_{\bowtie})(t) = lift(y - p)$. Instead, we compute $(lift_Y(R_{\bowtie}) \bowtie lift_{-P}(U))(t) = lift(y) \otimes lift(-p)$ without materializing $\mathcal{E}$. They are equivalent because $lift(\cdot)$ is addition-to-multiplication preserving. □

EXAMPLE 4. *The decision tree in Figure 2a has three leaf nodes: $(\sigma_{D \leq 1}, p=2.5), (\sigma_{D>1 \wedge C \leq 1}, p=1.5), (\sigma_{D>1 \wedge C>1}, p=2)$. The set of referenced attributes is $\mathbf{A} = \{C, D\}$. The update relation U is shown in Figure 2b. The semi-ring annotations lifted on $\mathcal{E} = Y - P$ over the materialized join are shown in Figure 2d. It is easy to verify that they are the same as those in $U \bowtie R_{\bowtie}$, without materialization $R_{\bowtie}$.*

4.2.2 Algorithmic Challenge.
Unfortunately, U can introduce cycles in the join graph over time (e.g., Figure 2c shows cycle $A \to C \to D$); whereas *Message Passing* requires acyclic join graphs. Standard hypertree decomposition [18, 47] removes cycles by joining the relations in a cycle, materializing their join result R' (e.g., $S[A, C] \bowtie T[A, D] \bowtie U[C, D]$), and replacing these relations in the join graph with R'. However, as the number of trees in model increases, the number of referenced attributes is likely to span the entire join graph—$|U|$ will thus converge to $|R_{\bowtie}|$.

To address this, we propose *Clustered Predicate Tree* (CPT): each boosted tree restricts split on features that can be pushed to the same fact table. To do so, we cluster relations such that within each cluster, a single fact table F maintains N-to-1 relationships with all other relations. Predicates in this cluster can be rewritten as semi-joins to the same fact table F (Section 4.1) and won't create cycles [18]. During training, while the root decision tree node can

---

[4]Additive inverse of $l.p$ exists because $\mathbb{R}$ is also a ring.

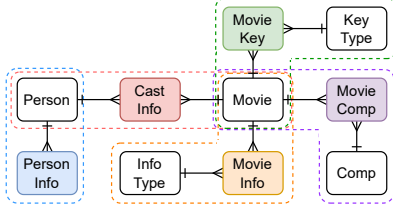[5]When $lift$ is applied to a column C, we denote it as $lift_C$.

Figure 3: Clusters for IMDB dataset. Each cluster is enclosed by dotted lines and its fact table is filled.

split on any feature, subsequent splits are confined to attributes within the same cluster. Although this might affect model accuracy, it allows efficient residual updates over join graphs, which would otherwise be untrainable using existing techniques.

> EXAMPLE 5. *Figure 3 shows the join graph of IMDB datasets [6], which was previously prohibitive to train gradient boosting due to the large join size ($R_{\bowtie}$ is >1TB). The five clusters are enclosed by dotted lines and the cluster's fact table is highlighted. If the current tree initially splits on Person's age (in Person Info), the rest of the tree can only split on attributes in Person or Person Info.*

5 JOINBOOST OVERVIEW AND OPTIMIZATIONS

We now present the system overview and optimizations.

5.1 JoinBoost Developer Interface

As described in Section 1, our design goals are portability, performance, and scalability. We evaluate performance and scalability in the experiments, so focus on portability below.

Portability: Our design deviates from prior factorized ML works [28, 48, 50, 68], which build (fast) custom execution engines. In contrast, JoinBoost is implemented as a Python library that transparently generates SQL queries to the user's DBMS backend (Figure 4).

API Compatibility: For usability, JoinBoost offers Python API similar to LightGBM and XGBoost, and returns models identical to LightGBM. In these libraries, users first define a training dataset and pass it, along with training parameters (e.g., objective, criteria, number of leaves), to a train() method. Likewise, JoinBoost users define the training dataset as a join graph (the relations and join conditions between them), providing a database connection, and specifying the features and target variable. If only relations are specified, JoinBoost infers the join graph that covers those relations from the database schema and raises an error if the graph is ambiguous (e.g., multiple foreign key references between relations) or requires cross-products. Finally, the user passes the training dataset and training parameters to train(). For consistency, JoinBoost accepts the same training parameters as LightGBM.

> EXAMPLE 6. *Figure 4 illustrates a simple example inspired by TPC-DS. The user creates a database connection, and initializes the training dataset as a join graph with relations* [sales, date], *join attribute* date_id, *features* X = [holiday, weekend] *and target variable* Y = net_profit. *Finally, the user chooses model parameters ($\{objective = regression\}$) and runs* train *over the training dataset.*

JoinBoost internally translates the ML algorithms into CREATE TABLE and SELECT SQL queries. Compared to in-DB systems like MADLib [38], JoinBoost generates pure SQL and does not require user-defined types or functions. This enables portability (criteria C2): JoinBoost runs on embedded databases, single-node databases, cloud data warehouses, and even Pandas and R dataframes [12].
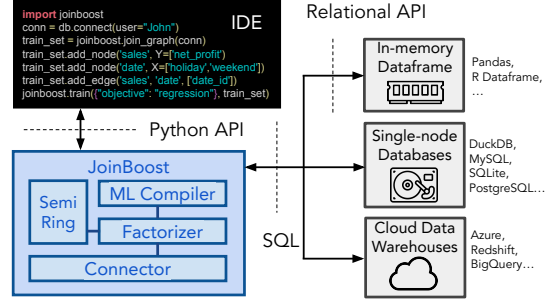

Figure 4: JoinBoost Architecture. JoinBoost translates its Python API calls into message passing algorithms that are executed as SQL queries on backend DBMSes and dataframes.

The compiler fully supports decision trees, random forests, and gradient boosting with all learning parameters that LightGBM supports. Currently, regression with *RMSE* objective supports galaxy schema with Clustered Predicate Trees; other objectives (e.g., *mae*, *huber*, *softmax*...) require snowflake schema. We are actively extending capabilities to support pruning, dropout, and early stopping, which build on the techniques in the preceding sections.

One usability challenge is that JoinBoost end users are typically domain scientists who lack expertise in data access and schema, commonly handled by data engineers. While addressing this is beyond the scope, we highlight the recent trend of "semantic layer" [9, 10, 15, 26] as a means to bridge the knowledge gap. In the "semantic layer," data engineers employ SQL-based recipes to transform and preprocess raw data into tables containing meaningful attributes to domain scientists, and can be queried by JoinBoost using SQL. However, current "semantic layer" still materializes costly joins, given most downstream applications anticipate a single table. Integrating JoinBoost with semantic layers to avoid join materialization is a potential avenue for future work.

Safety: Training shall never modify user data. To achieve that, JoinBoost creates temporary tables in a specified namespace or with a unique prefix. By default, JoinBoost deletes these tables after training, but users can keep them for provenance or debugging.

5.2 Architecture Overview

The *ML Compiler* runs the ML logics (Algorithm 1) to train tree-based models in Python and translates the most computational intensive L14, which identifies the best split across features for tree nodes, into SQL queries to be executed by DBMS. At this stage, the compiler treats the join graph as a single "wide" table, with the SQL query operating on this table, and factorization is applied in a later step. After training, it returns a reference to the trained model.

The *Semi-ring Library* stores semi-ring definitions and translates math expressions in the compiler-generated queries ($\times$, $+$, $lift$) into SQL aggregation functions. $lift(R)$ creates a copy of a base relation R that contains an additional attribute for each component in the semi-ring (e.g., c, s, q in the variance semi-ring). This also ensures that any update in-place will not modify user data. In addition to the variance (for regression) and the class count (for classification) semi-rings, JoinBoost implements semi-rings for a wide range of popular objectives; it supports *rmse* for snowflake and galaxy schemas, and *mae, huber loss, fair loss, log loss, softmax* and more for snowflake schemas (see Appendix B for full list).

The *Factorizer* decomposes each aggregation query into message passing and absorption queries. It also materializes each message as a database table, and re-uses them when possible. After choosing a node split, the factorizer keeps messages that can be reused in descendent nodes (Section 5.5.1) and drops the rest.

Finally, the *Connector* takes our internal SQL representation, and translates them into the appropriate SQL string or dataframe API calls. Although DBMSes are notorious for incompatible SQL variants, `JoinBoost` only uses a subset of SQL that is generally consistent across vendors. For instance, it generates standard non-nested SPJA queries with simple algebra expressions.

5.3 Residual Updates Logical Optimization

Although correct, the residual update technique in Section 4 is still expensive to implement naively. For simplicity, if we assume a single join attribute A between the fact table F and update table U, and the variance semiring, the SQL query would be:

```
CREATE TABLE F_updated AS
    SELECT F.c*U.c AS c, F.s*U.c+U.s*F.c AS s,
           F.q*U.c+U.q*F.c+2*F.s*U.s AS q, ...
      FROM F JOIN U ON A
```

where c, s, q are semiring components, and the remaining columns in F are copied over (shown as . . .). Unfortunately, this is >50× slower than `LightGBM`'s residual update procedure (see experiments below), because U can potentially be as large as the materialized $R_{\bowtie}$. To this end, we present an optimization for snowflake and galaxy schemas that completely avoids materializing U as well as $F \bowtie U$.

5.3.1 Semi-join Update Optimization. We directly UPDATE F's semi-ring annotations. Let us start with a snowflake schema. Each decision tree leaf l logically corresponds to a separate join graph containing a set of messages (Figure 6), with $l.\sigma$ as its predicate and $l.p$ as its prediction. Using the semi-join optimization (Section 4), we translate predicates over $R_{\bowtie}$ (e.g., $l.\sigma$) into semi-joins between F and relevant incoming messages $\mathcal{M}$. A message $m_i \in \mathcal{M}$ is relevant if it is along a join path from a relation containing an attribute in $l.\sigma$ to F. For each leaf node l, we execute the following query where $F.a_i$ is the join attribute with its relevant incoming message m_i:

```
UPDATE F SET c * 1 AS c, s - l.p*c AS s, q + l.p*l.p*c - 2*s*l.p AS q
      WHERE F.ai IN (SELECT ai IN mi) AND ...
```

In some databases, updates in place can be very slow. Thus an alternative is to create a new fact table with the updated semi-ring annotations. Let l_j be the jth leaf in the decision tree, and $m_{i,j}, a_{i,j}$ be the ith message and its join attribute with F in l_j's join graph:

```
CREATE TABLE F_updated AS SELECT
    CASE WHEN F.a_ij IN (SELECT a_ij FROM m_ij) AND ... THEN s - l_j.p*c
         WHEN ... // Other leaves
      END AS s, ...  // other semi-ring components
       ...  // copy other attributes in F
    FROM F
```

The same ideas apply to galaxy schemas, where F corresponds to the fact table of the current tree's cluster. Further, we show in the technical report [13] that c and q are not necessary to materialize; thus only s is needed for the variance semi-ring.

5.3.2 Pilot Study. When should we perform in-place updates as compared to creating new tables? On which DBMSes? We now report a microbenchmark to understand the performance trade-offs, and use them to motivate a new optimization. We used an Azure VM, with 16 cores, 128 GB memory, and an SSD.
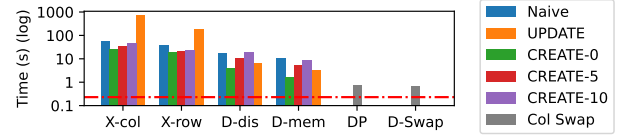

Figure 5: Residual update time (log) for different DBMSes using different methods. The red horizontal line is the residual update time for `LightGBM`. Column swap (DP, D-Swap) achieves competitive residual update performance.

Workloads. We create a synthetic fact table $F(s, d, c_1, ..., c_k)$ with $100M$ rows to simulate residual updates in a decision tree with 8 leaves. s is the semi-ring column to update, $d \in [1, 10K]$ is the join key, and c_k are simply extra columns that would need to be duplicated in a new table. For the ith leaf node, its prediction is a random float, and we construct its semi-join message $m_i(d)$ to contain all values in $(1250 \times (i - 1), 1250 \times i]$.

Methods. We evaluate three approaches. Naive materializes Update Relation U, then re-create fact table: $F' = F \bowtie_A U$ as discussed in Section 4. SET and CREATE use the update-in-place and create table optimizations in the preceding subsubsection. CREATE-k denotes the number of extra columns in F, where $k \in \{0, 5, 10\}$; we set $k = 0$ for Naive, and k does not affect SET.

DBMSes. We evaluate two systems. DBMS-X is a popular commercial RDBMS that supports both column-oriented (X-col) and row-oriented (X-row) storage and query processing. DBMS-X is disk-based only, and we set the isolation and recovery to the lowest level (read uncommitted and minimum logging). DuckDB [64] is a popular embedded column-oriented OLAP DBMS and is highly performant [14]. DuckDB has disk-based (D-disk) and memory-based (D-mem) modes. As a reference, we also use `LightGBM` to train 1 iteration of gradient Boosting with the same training settings, and report residual update time.

Experiment Results. Figure 5 shows that Naive incurs high materialization and join costs. CREATE is $\sim 2\times$ faster for DBMS-X and $\sim 4\times$ faster for DuckDB, but its cost grows linearly with k. SET mainly depends on the DBMS—it is prohibitive for DBMS-X, but more efficient than CREATE when $k > 5$ for DuckDB. All DBMS approaches take >3s for updating residuals. In contrast, `LightGBM` stores the target variable in a C++ array and performs parallel writes; its residual update takes ~0.2s. These poor results are due to four main factors.

- **Compression:** CREATE in X-col incurs high compression costs: the database is $1GB$ in X-col as compared to $2.6GB$ for DuckDB. This also penalizes SET due to decompression.
- **Write-ahead Log (WAL)** introduces costly disk writes.
- **Concurrency Control (CC):** In-memory DuckDB doesn't use WAL, but incurs MVCC [56] overheads, including versioning, and logging for undo and validation.
- **Implementation:** DuckDB's update is currently single-threaded.

5.4 Residual Updates Physical Optimization

Logical rewrites are effective but still much slower than `LightGBM`'s residual updates, even when existing CC mechanisms are lowered. We observe that `JoinBoost` does not need durability and concurrency control, since it writes to private tables, performs application-level concurrency control (Section 5.5), and can simply re-run upon
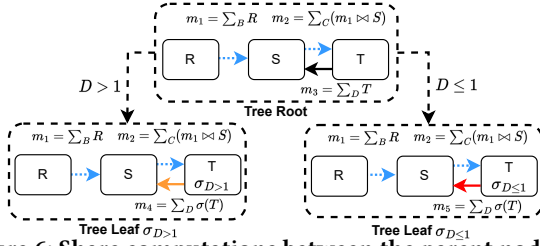
Figure 6: Share computations between the parent node and children nodes after the split. Dotted messages are shared.

failure. Compression is also unnecessary for the heavily updated columns. Unfortunately, WAL, CC, and compression are deeply integrated into DBMS designs, and nontrivial to fully disable[6].

We note that columnar engines are well suited to avoid these costs by adding the new residual column as a projection [74], but is not generally supported (e.g., by DuckDB, DBMS-X). We thus evaluate its benefits using three methods: one that uses DuckDB's existing APIs to swap columns in Pandas dataframe, one that adds "column swapping" to DuckDB, and one that simulates it in DBMS-X.

The first solution, called DP (DuckDB + Pandas), uses the existing DuckDB Relational API [12] to directly access Pandas dataframes [60] – an in-memory data structure that stores columns in contiguous uncompressed C arrays. Internally, it uses a custom scan operator for Pandas's matrix format, and DuckDB's executor for the rest of the query. We store the fact table F as a dataframe, and the remaining tables in DuckDB, and join them via the relational API during training. For residual update, we submit a query over Pandas and native relations to compute the semi-ring annotations for the updated residuals, and store the result in a NumPy array. Then using Pandas, we replace the old column in F with the new NumPy array (a pointer swap). This is fast because it avoids WAL or CC overhead, and reduces residual updates to $0.72s$–competitive with LightGBM (Figure 5). The drawback is that the scan overhead slows down the join-aggregation query, and increases training time by $\sim 1.6\times$ as compared to querying only native relations (Section 6.3).

The second approach, D-Swap, modifies DuckDB internals slightly (<100 LOC) to support pointer-based column swapping between two DuckDB tables. DuckDB stores tables in row groups, with each group containing pointers to the column data. D-Swap iterates through the row groups of the two relations and swaps the column pointers. Such column swap is a schema-level modification, so it is very fast and side-steps decompression, CC, and WAL overheads. It achieves similar residual update performance to DP (Figure 5) without degrading aggregations (Section 6.3).

Finally, to extend beyond DuckDB, we also simulate column swapping's impact on DBMS-X in Section 6.3, and see a potential $15\times$ improvement. We believe that implementing such column swap in closed-source DBMSes is both feasible and effective for residual updates. We encourage columnar DBMS developers to incorporate this operation for efficient In-DB gradient boosting.

5.5 Optimizations and Features

We implement various optimizations and features in JoinBoost for performance and usability. Due to space limit, we only discuss the most critical ones here, while the rest can be found in Appendix D.

---

[6]Verified with DuckDB and DBMS-X developers.

5.5.1 Message Sharing Among nodes. Previous factorized ML algorithms [50, 68] rely on batch optimization of aggregation queries, suitable for models with closed-form solutions (e.g., ridge regression [68]). However, tree-based model training is iterative: queries for child nodes depend on the parent node's split, making batching ahead impossible. Consequently, previous algorithms either miss work-sharing opportunities between tree nodes or batch all possible splits, which is impractical due to exponential growth in tree depth. Our key observation is that messages as intermediate results can also be shared among tree nodes. Consider the example:

EXAMPLE 7. *Following Example 3, suppose the best root split is $\sigma_{D>1}$. The split applies σ and $\neg\sigma$ to T (as it contains D), creating two leaf nodes with relations: $\sigma(R_{\bowtie})$ and $\neg\sigma(R_{\bowtie})$. Then, for each leaf node, we compute the batch of aggregation queries and identify the next split through Message Passing. As shown in Figure 6, messages (dotted blue) along the path $R \to S \to T$ are the same for both leaf and root nodes. We only need to recompute message $m_{T\to S}$ in each tree leaf, skipping $m_{S\to R}$ as R's attributes aren't used in this model.*

In general, after a split on R_i, all messages along the path to R_i can be re-used. This is orthogonal to prior batch optimization work [68] as we can cache and reuse messages after batching for future nodes, further improving batch optimization by $>3\times$ (Section 6.4).

5.5.2 Sampling for Random Forest. A random forest model simply trains multiple decision trees over random samples of the training data and features, and aggregates (e.g., averages) their predictions during inference. Feature sampling can be easily implemented by using a random subset of $\mathbf{X}' \subseteq \mathbf{X}$ features for Algorithm 1. The main challenge is efficiently sampling over non-materialized $R_{\bowtie}$: Naively sampling each relation is (1) not uniform and independent, and (2) may produce non-joinable and hence empty samples.

To address these, we use previous ancestral sampling [25, 70, 79]: Given $R_{\bowtie}=R_1 \bowtie R_2 ... \bowtie R_n$ with total *COUNT C* (as a constant), ancestral sampling treats $R_{\bowtie}$ as the full probability table, with each tuple's probability $1/C$. Then, the marginal probability over a set of attributes $\mathbf{A} \subseteq S_{R_{\bowtie}}$ is $P(\mathbf{A}) = \gamma_{\mathbf{A},count(*)/C}(R_{\bowtie})$. Ancestral sampling starts from a root relation (e.g., R_1) in the join graph and draws weighted sample t based on its marginal probability (e.g., $P(S_{R_1})$); $P(S_{R_1})$ requires a *COUNT* semi-ring aggregation query that can be computed without materializing $R_{\bowtie}$ (Section 3.1). It then traverses the join graph to the next relation (e.g., R_2 with the join key $\mathcal{J} = S_{R_1} \cap S_{R_2}$), filtering R_2 by tuples that join with t ($\sigma_{\mathcal{J}=\pi_{\mathcal{J}}(t)}(R_2)$), and recursively apply sampling based on the conditional probability $P(S_{R_2} - \mathcal{J}|\mathcal{J}) = P(S_{R_2})/P(\mathcal{J})$. During sampling, the expensive queries are $P(S_{R_1}), ...P(S_{R_n})$, which can be accelerated through caching messages (Section 3.3).

Minor Optimizations. First, we coalesce the messages for *COUNT* query with those for the tree criterion. For instance, the c element in Variance Semi-ring captures the *COUNT* statistics. Second, for snowflake schemas where the fact table has N-to-1 relationships with the rest of the tables, we sample the fact table directly [76].

5.5.3 Inter-query Parallelism. Parallelism is widely used in ML libraries like LightGBM, which implements parallelized sorting, aggregation, residual updates, and split candidate evaluations. For JoinBoost, most DBMSes offer intra-query parallelism, but there can be diminishing returns for individual queries or operations.
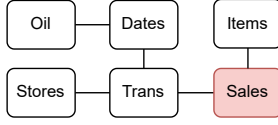
Figure 7: Favorita schema. `Sales` is the fact table.

Thus, `JoinBoost` also aggressively parallelizes across trees, leaf nodes, candidate splits, and messages. However, there are dependencies between these queries: a message's query relies on upstream messages, absorption depends on incoming messages, tree nodes depend on ancestor node queries, and gradient boosting trees rely on preceding trees. To handle this, `JoinBoost` employs a simple scheduler. Each query Q tracks its dependent queries, and when Q finishes, it sets the ready bit for these dependencies. If all dependencies for a query are set, it's added to a FIFO run queue. Empirically, 4 threads work best for intra-query parallelism, while the rest are used for inter-query parallelism. This reduces gradient boosting training time by 28% and random forest by 35% (Appendix C).

6 EXPERIMENTS

We focus on the fastest alternative: ML libraries (XGBoost, `LightGBM`, `Sklearn`). We start with a single-node setting, and then evaluate scalability to the number of features, types of joins, and database size on multiple nodes. Finally, we compare with factorized (`LMFAO`) and non-factorized (`MADLib`) in-DB ML techniques.

Datasets. We primarily report results using the Favorita [7] dataset used in prior factorized ML work [68, 69] (Figure 7). `Sales` is the fact table (2.7GB, 80M rows), and has N-to-1 relationships with the other dimensions ($<2MB$ each). There are 13 features. We report TPC-DS results for scalability experiments and use IMDB for galaxy schema experiments; additional TPC-H/DS results are in the Appendix C.

Preprocess. Although Favorita and TPC-H/DS are standard benchmarks in prior factorized learning evaluations [68], their features are non-predictive and lead to highly unbalanced trees. This artificially favors `JoinBoost` since all but one leaf node in the decision tree would contain very few records, and thus take negligible training time—the performance differences with other systems are fully dominated by join materialization. To ensure balanced trees and fair comparison, we impute one feature attribute in each of the 5 dimension tables with random integers drawn from [1, 1000]. Then we impute the target variable as sum of transformed features[7]. Finally, we dictionary encode strings into 32-bit unsigned integers [17, 65] to avoid parsing errors in ML libraries like `LightGBM`.

Models. We evaluate decision tree, random forest, and gradient boosting. `JoinBoost` is intended to complement other DBMS workloads, so all experiments start with data persistent on disk but not in memory. We assume by default that data are already persisted in the disk-based DBMSes (DBMS-X and disk-based `DuckDB`). We report the end-to-end training time for decision tree of max depth 10, and vary the number of trees (iterations) in the random forest and gradient boosting. For `JoinBoost`, the main cost is from DBMSes, and the Python codes introduce negligible (<0.1s) overhead.

Methods. We evaluate `JoinBoost` with different DBMS backends. DBMS-X (X-col and X-row for column and row-oriented storage and execution engines) and `DuckDB-disk` (D-disk) are disk-based and

directly execute queries on the base DBMS, whereas memory-based `DuckDB` (D-mem) first loads the DBMS from disk. DP refers to the disk-based `DuckDB` backend using `Pandas` updates through the `DuckDB`'s relational API, and D-Swap refers to the modified memory-based `DuckDB` for efficient residual updates (Section 5.4). By default, we use D-Swap as the backend as it has the best performance (Section 6.3). ML libraries[8] (`LightGBM`, XGBoost, `Sklearn`) expect a single CSV as input, so incur the cost to materialize and export the join result ($\sim 7GB$ for Favorita), load the CSV, and train the model. `DuckDB` joins and exports the data faster than DBMS-X, so we report its numbers.

Hardware: We use Azure VM: 16 cores, 128 GB memory, and SSD.

6.1 Comparison With ML Libraries

We compare with SOTA ML libraries for training tree-based models (`LightGBM` [48], XGBoost [28] and `Sklearn` [63]). `Sklearn` implements the standard and histogram-based gradient boosting with algorithms similar to `LightGBM`, so we report both implementations. We set the number of bins to 1000 for `LightGBM` and XGBoost, and 255 for `Sklearn` (its limit). By default, we train gradient boosting and random forest with best-first growth, with a maximum of 8 leaves per tree. The gradient boosting learning rate is 0.1. The random forest data sampling rate without replacement is 10%, and feature sampling rate is 80%. We report up to 100 iterations. The 0th iteration reports the join materialization, export, and load costs.

Figure 8a shows random forest results. `JoinBoost` is $\sim 3\times$ faster than `LightGBM` by avoiding materialization and export costs (dotted black line), and loading costs; it also parallelizes across trees. In fact, `JoinBoost` finishes 100 iterations before the export is done. `Sklearn` also parallelizes across trees, but is so slow that we terminate after 32 iterations. The final model error ($rmse$) is nearly identical (~ 2350) for `JoinBoost`, `LightGBM` and XGBoost.

Figure 8b shows gradient boosting results. `JoinBoost` is $\sim 1.1\times$ faster than `LightGBM`, and is $\sim 1.2\times$ faster than XGBoost by avoiding materialization and export costs. Figure 8c shows the model $rmse$. `JoinBoost` and `LightGBM` have equivalent $rmse$ as both employ the same algorithm, while the final $rmse$ is similar across all. Note that the models begin to converge ~ 60 iterations; `JoinBoost` converges by the time `LightGBM` and XGBoost have loaded their data.

To interpret the `JoinBoost` cost, we zoom in on the 1st iteration of `JoinBoost` gradient boosting in Figure 9, which displays the total number of queries for passing messages (blue) and finding the best split (orange) along with a histogram for the query execution time distribution. With a tree of 8 leaves and 15 nodes, there are $270 = 15 \times 18$ (number of features) queries for feature split, and $75 = 15 \times 5$ (number of join edges) queries for message passing, as expected. All feature split queries are efficient, taking $< 10ms$. The performance bottlenecks are the queries passing messages from the fact table, which require join-aggregation, result materialization, and take $> 200ms$. This highlights the importance of aggressively reusing messages across nodes (Section 5.5.1).

6.2 Scalability

We now study scalability to # features, DB size, and join complexity. **# Features.** We train gradient boosting using Favorita and vary the

---

[7]Favorita applies: $y = f_{item} log(f_{items}) + log(f_{oil}) - 10 f_{dates} - 10 f_{stores} + f_{trans}^2$

[8]Python version `LightGBM` and XGBoost have memory issues [8] so we use CLI version.
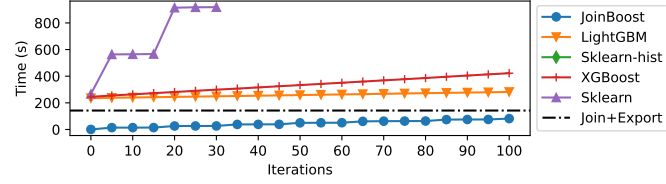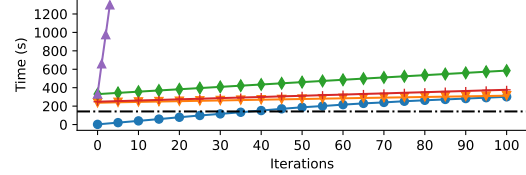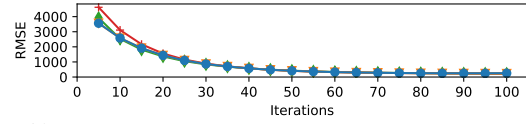
(a) Random forest training time. `JoinBoost` is ~3× faster than `LightGBM`.

(b) Gradient boosting Training time. `JoinBoost` is ~1.1× faster.

(c) Gradient boosting Accuracy. The final *rmse* is nearly identical.

Figure 8: Gradient boosting and random forest training time and accuracy on Favorita compared to ML libraries when the materialized join fits into memory on a single node.
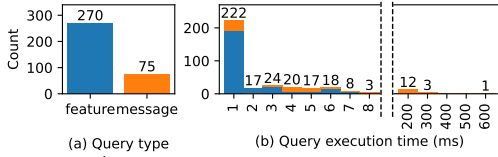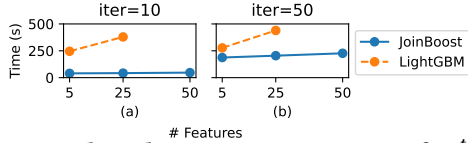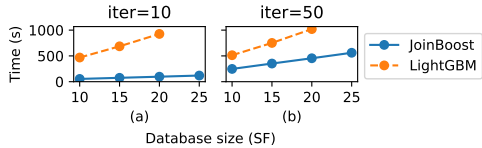

Figure 9: The 1st iteration of `JoinBoost` gradient boosting. (a) The number of queries for passing messages and finding the best split. (b) The histogram of query execution time (ms).

Figure 10: Gradient boosting training time of 10th (a) and 50th (b) iteration when varying # of imputed features (x-axis). `LightGBM` runs out of memory when imputing 50 features.

Figure 11: Gradient boosting training time of 10th (a) and 50th (b) iteration. The X-axis varies TPC-DS SF (database size). `LightGBM` runs out of memory when SF=25.
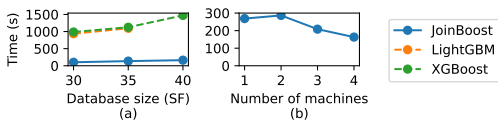
Figure 12: Gradient boosting training time of 10 iterations (a) varying SF on 4 machines (b) varying # of machines for SF=40. `LightGBM` runs out of memory at SF=40 even on 4 machines.
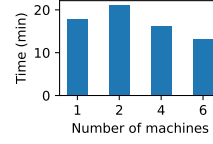

Figure 13: Decision Tree Training Time over TPC-DS (SF=1000) in Data Warehouse. Increasing the number of machines to 4 (6) reduces the training time by 10% (25%).

number of features from 5 to 50, and report training time at the 10th and 50th iterations. Figure 10 shows that `LightGBM` slows by >1.5× with 25 features, and runs out of memory (125GB) at 50 features. `XGBoost` supports out-of-core training, but took~4000s to train 50 features for 10 iterations, and wrote ~80GB of intermediate results to disk (not plotted). `JoinBoost` scales linearly with >10× lower slope, and scales to out-of-core execution thanks to the DBMS.

Single-node Scalability. Favorita is a fixed dataset, so we use TPC-DS (145 features) to scale the database ($SF \in [10, 25]$). Figure 11 shows that both systems scale linearly, but `JoinBoost` has a lower slope (~10× lower at 10th iteration, and ~2× lower at 50th). `LightGBM` runs out of memory at $SF = 25$.

Multi-node Scalability. We use the Dask version [66] of `LightGBM` and `XGBoost`, and Dask-SQL for `JoinBoost` to train gradient boosting on multiple machines for 10 iterations on TPC-DS $SF \in [30, 40]$. We use 4 n1-highmem-16 GCP instances (16 vCPUs, 104 GB RAM each) with data replicated across all instances; at least 4 machines are needed for `LightGBM` to run on $SF = 30$ due to large join sizes and inefficient memory usage [8]. Figure 12 (a) shows that, on 4 machines, all systems scale linearly , but `JoinBoost` is >9× faster with a ~5× lower slope. For SF=40, `LightGBM` runs out of memory even on 4 machines. In contrast, Figure 12 (b) shows that for SF=40, `JoinBoost` can train on a single machine, outperforms `XGBoost` (using 4 machines), and speeds up with more machines.

Cloud-warehouse Scalability. We use a cloud data warehouse `DW-X` to train a decision tree with max depth 3 over TPC-DS SF=1000, and study multi-machine scalability. Each machine has 74 cores, 300GB of memory, and SSDs. We replicate dimensional tables across the machines, and hash-partition the fact table. Figure 13 shows that 2 machines introduce a shuffle stage that slows training, and increasing to 4 (6) machines reduces training by 10% (25%).

Galaxy Schemas. Galaxy schemas have N-to-N relationships that are prohibitive to materialize. We use clustered predicate trees (Section 4.2.2) to train gradient boosting on the IMDB dataset (Figure 3). `Cast_Info` is ~1GB and the total DB is 1.2GB. `JoinBoost` scales linearly with the number of iterations (Figure 14); it trains one tree and updates residuals in each cluster's fact table within ~5s. `LightGBM` cannot run because the join result is >1TB.

6.3 Effect of DBMSes

We now use Favorita to train 1 iteration of gradient boosting, and compare train and update costs for different DBMSes (`DBMS-X` and `DuckDB`). To study the potential benefits of column swap (Section 5.4) in commercial DBMSes, `X-Swap*` reports the theoretical cost using the time to create the updated column as a new table (since column swap is then "free").
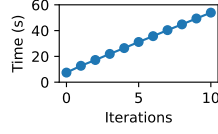
Figure 14: Gradient Boosting Training over IMDB dataset with a galaxy schema. Each tree takes ∼5s. ML libraries do not run because the join is too large to materialize.
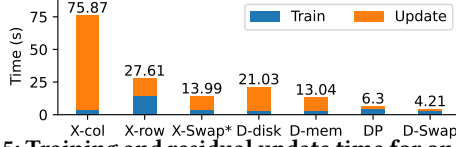

Figure 15: Training and residual update time for one decision tree on Favorita for different DBMSes. `X-Swap*` is the theoretical runtime by simulating column swaps.
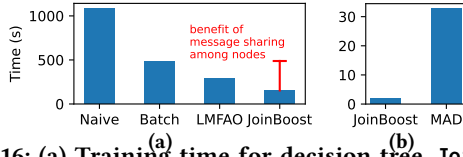

Figure 16: (a) Training time for decision tree. `JoinBoost`'s caching and work sharing improve over `LMFAO`, naive materialization and query batching variants. (c) Training time for decision tree. `JoinBoost` is ∼16× faster than `MADLib`.

Figure 15 breaks down train and update costs. Decision trees and random forests only require training (blue bar), which is dominated by columnar execution: `X-col` and `DuckDB` take $3.2 − 3.9s$ versus $14.5s$ for `X-row`. Gradient boosting introduces high update costs across all of the baseline DBMSes. We see that using Pandas to perform the update (DP) reduces residual updates by ∼15× ($17.8s{\rightarrow}1.2s$), but slows training by 60% ($3.2s{\rightarrow}5.1s$) due to `DuckDB`-Pandas interop overhead. `D-opt` implements column swapping inside DuckDB and improves training. We also see that adding column swapping to `DBMS-X` (`X-Swap*`) leads to respectable gradient boosting performance (∼3× of `D-Swap`) but can benefit from its out-of-memory and multi-node scalability features.

6.4 Comparison With In-DB ML Techniques

We first compare with the dominant factorized approach (`LMFAO` [68]), followed by the non-factorized approach (`MADLib`).

Factorized ML. We first compare with `LMFAO` [68], which supports decision trees (but not gradient boosting or random forests). We train a decision tree (max depth=10) with best-first growth; the trained tree is balanced and has 1024 leaves. Via correspondence, the `LMFAO` authors shared a version that compiles a program for the queries used to split the root tree node, and reuses it for growing the rest of the tree. We set the highest optimization level for `LMFAO`, and exclude the time for query compilation (∼15s) and data loading. To separate algorithmic vs implementation differences, we implement two variations of `JoinBoost`. `Naive` materializes the join result without factorized ML. `Batch` implements LMFAO's core logical optimizations (Multi Root, Aggregate Push-down and Merge View) for decision tree; this corresponds to message passing with message re-use within, but not between tree nodes.

Figure 16a reports the training time. Even with the custom engine and specialized optimizations, `LMFAO` is still ∼1.9× slower than `JoinBoost` due to the lack of message caching (Section 5.5.1). By

eliminating the implementation differences, `Batch` demonstrates that message caching improves training by ∼3×. The improvement is because half the messages across tree nodes are cached, and the sizes of the cached messages tend to be much larger; the intuition is that the messages outgoing from the relation containing the split attribute will be smaller, since the split predicate has been applied. In theory, two-pass semi-join reduction [18, 59] could reduce message sizes, but the high overheads outweigh benefits [17]. `Batch` is ∼2× faster than `Naive` due to factorization and shared work.

Non-Factorized ML. `MADLib` [38] is a PostgreSQL extension that supports ML using user-defined types and functions. `MADLib` doesn't apply factorized ML, so the join has to be materialized. `MADLib` times out after 1 hour when training a decision tree model (max depth=10) on the full Favorita dataset, so we reduced the training data size to 10k rows; for `JoinBoost`, we reduced the fact table to 10k rows. Figure 16b shows that `JoinBoost` is ∼16× faster than `MADLib`. This reproduces prior findings [68], and is likely due to the lack of factorized ML and an inefficient implementation.

7 LIMITATIONS AND FUTURE WORK

Limitations. `JoinBoost` has some limitations that may prevent its widespread use. For usability, it requires users (e.g., data scientists) to specify schema and join conditions if they cannot be inferred automatically. In terms of algorithm, `JoinBoost` currently supports the most widely used criterion, *rmse*, for galaxy schema using *Clustered Predicate Trees* (CPT). However, efficiently factorizing residual updates for other criteria remains challenging (Section 4).

Future work. In the future, our goals include enhancing `JoinBoost`'s usability by integrating it with "semantic layers"[9, 10, 15, 26] that transform raw data into tables with meaningful attributes for domain scientists and extending them for join graphs. To support more criteria, one promising future direction is to apply polynomial approximations [43]: variance semi-ring computes 2nd order monomials $(1, y, y^2)$, which can be extended to higher order $(1, y, y^2, ..., y^k)$ with tailored multiplication operation to both update monomials and preserve addition-to-multiplication property. Monomials can approximate various criteria by Taylor expansion.

Additionally, applying `JoinBoost` to GPU-based DBMSes [40] is of interest, as GPU memory size is notoriously limited [20] especially for join, and `JoinBoost` may address this. We also aim to explore data augmentation search [29], which joins training data with additional features and evaluates model (random forest) improvement. `JoinBoost`'s scalability and portability can be leveraged for more efficient augmentation search in enterprise data warehouses. Lastly, examining the explainability of gradient boosting, particularly recent developments in causal inference [21] with potential interactions between causal and join graphs, will offer insights into gradient boosting decision-making processes.

8 CONCLUSION

`JoinBoost` is the first In-DB factorized ML system for tree models (decision trees, random forests, and gradient boosting) with only SQL. `JoinBoost` is comparable or faster than the SOTA `LightGBM` ML library on in-memory datasets, but scales well beyond `LightGBM`'s capabilities in terms of # of features, database size, and join graph complexity. `JoinBoost` exposes a Python API that mimics `LightGBM`'s API and is portable to any DBMS and dataframe library.

REFERENCES

[1] [n.d.]. Azure Machine Learning documentation. https://learn.microsoft.com/en-us/azure/machine-learning/.

[2] [n.d.]. Personal-Data-Protection-Act. https://www.pdpc.gov.sg/Overview-of-PDPA/The-Legislation/Personal-Data-Protection-Act.

[3] [n.d.]. Snowflake Machine Learning Platforms. https://www.snowflake.com/guides/machine-learning-platforms.

[4] [n.d.]. Using machine learning in Amazon Redshift. https://docs.amazonaws.cn/en_us/redshift/latest/dg/machine_learning.html.

[5] 2006. Updates through views: A new hope. In *22nd International Conference on Data Engineering (ICDE'06)*. IEEE, 2–2.

[6] 2013. IMDB. https://www.imdb.com/interfaces/.

[7] 2017. Corporación Favorita Grocery Sales Forecasting. https://www.kaggle.com/c/favorita-grocery-sales-forecasting.

[8] 2017. Lightgbm memory explodes in start train. https://github.com/microsoft/LightGBM/issues/1032.

[9] 2020. Looker data modeling. https://www.looker.com/platform/data-modeling/.

[10] 2020. The Tableau Data Model. https://help.tableau.com/current/online/en-us/datasource_datamodel.htm.

[11] 2021. 2021 Kaggle Data Science and Machine Learning Survey. https://www.kaggle.com/code/paultimothymooney/2021-kaggle-data-science-machine-learning-survey/notebook.

[12] 2021. Client APIs Overview. https://duckdb.org/docs/api/overview.

[13] 2021. (Technical Report) JoinBoost: In-Database Tree-Models In Action. https://anonymous.4open.science/r/JoinBoost-FBC4/technical/JoinBoost_tech.pdf.

[14] 2022. ClickBench: a Benchmark For Analytical Databases. https://benchmark.clickhouse.com/.

[15] 2023. The dbt Semantic Layer. https://www.getdbt.com/product/semantic-layer/.

[16] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/ Software available from tensorflow.org.

[17] Christopher R Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. 2017. Emptyheaded: A relational engine for graph processing. *ACM Transactions on Database Systems (TODS)* 42, 4 (2017), 1–44.

[18] Mahmoud Abo Khamis, Hung Q Ngo, and Atri Rudra. 2016. FAQ: questions asked frequently. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. 13–28.

[19] Andreea Anghel, Nikolaos Papandreou, Thomas Parnell, Alessandro De Palma, and Haralampos Pozidis. 2018. Benchmarking and optimization of gradient boosting decision tree algorithms. *arXiv preprint arXiv:1809.04559* (2018).

[20] Yuki Asada, Victor Fu, Apurva Gandhi, Advitya Gemawat, Lihao Zhang, Dong He, Vivek Gupta, Ehi Nosakhare, Dalitso Banda, Rathijit Sen, et al. 2022. Share the tensor tea: how databases can leverage the machine learning ecosystem. *arXiv preprint arXiv:2209.04579* (2022).

[21] Edvard Bakhitov and Amandeep Singh. 2022. Causal gradient boosting: Boosted instrumental variable regression. In *Proceedings of the 23rd ACM Conference on Economics and Computation*. 604–605.

[22] Matthias Boehm, Michael W Dusenberry, Deron Eriksson, Alexandre V Evfimievski, Faraz Makari Manshadi, Niketan Pansare, Berthold Reinwald, Frederick R Reiss, Prithviraj Sen, Arvind C Surve, et al. 2016. Systemml: Declarative machine learning on spark. *Proceedings of the VLDB Endowment* 9, 13 (2016), 1425–1436.

[23] Leo Breiman. 2001. Random forests. *Machine learning* 45, 1 (2001), 5–32.

[24] Leo Breiman, Jerome H Friedman, Richard A Olshen, and Charles J Stone. 2017. *Classification and regression trees*. Routledge.

[25] Bishop PRML Ch and Alireza Ghane. 1993. Sampling Methods. (1993).

[26] Damianos Chatziantoniou and Verena Kantere. 2020. Data Virtual Machines: Data-Driven Conceptual Modeling of Big Data Infrastructures.. In *EDBT/ICDT Workshops*.

[27] Lingjiao Chen, Arun Kumar, Jeffrey Naughton, and Jignesh M Patel. 2016. Towards linear algebra over normalized data. *arXiv preprint arXiv:1612.07448* (2016).

[28] Tianqi Chen and Carlos Guestrin. 2016. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*. 785–794.

[29] Nadiia Chepurko, Ryan Marcus, Emanuel Zgraggen, Raul Castro Fernandez, Tim Kraska, and David Karger. 2020. ARDA: automatic relational data augmentation for machine learning. *arXiv preprint arXiv:2003.09758* (2020).

[30] Francois Chollet et al. 2015. *Keras*. https://github.com/fchollet/keras

[31] Conor Cunningham, César A Galindo-Legaria, and Goetz Graefe. 2004. PIVOT and UNPIVOT: Optimization and Execution Strategies in an RDBMS. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*. 998–1009.

[32] Ryan Curtin, Benjamin Moseley, Hung Ngo, XuanLong Nguyen, Dan Olteanu, and Maximilian Schleich. 2020. Rk-means: Fast clustering for relational data. In *International Conference on Artificial Intelligence and Statistics*. PMLR, 2742–2752.

[33] Xixuan Feng, Arun Kumar, Benjamin Recht, and Christopher Ré. 2012. Towards a unified architecture for in-RDBMS analytics. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. 325–336.

[34] Jerome H Friedman. 2002. Stochastic gradient boosting. *Computational statistics & data analysis* 38, 4 (2002), 367–378.

[35] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. 1997. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data mining and knowledge discovery* 1, 1 (1997), 29–53.

[36] Todd J Green, Grigoris Karvounarakis, and Val Tannen. 2007. Provenance semirings. In *Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. 31–40.

[37] Léo Grinsztajn, Edouard Oyallon, and Gaël Varoquaux. 2022. Why do tree-based models still outperform deep learning on tabular data? *arXiv preprint arXiv:2207.08815* (2022).

[38] Joe Hellerstein, Christopher 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. *arXiv preprint arXiv:1208.4165* (2012).

[39] Yu-Ching Hu, Yuliang Li, and Hung-Wei Tseng. 2021. TCUDB: Accelerating Database with Tensor Processors. *arXiv preprint arXiv:2112.07552* (2021).

[40] Zezhou Huang, Pavan Kalyan Damalapati, Rathijit Sen, and Eugene Wu. 2023. Random Forests over Normalized Data in CPU-GPU DBMSes. In *Data Management on New Hardware*.

[41] Zezhou Huang, Pavan Kalyan Damalapati, and Eugene Wu. 2023. Aggregation Consistency Errors in Semantic Layers and How to Avoid Them. (2023).

[42] Zezhou Huang and Eugene Wu. 2022. Calibration: A Simple Trick for Wide-table Delta Analytics. arXiv:2210.03851 [cs.DB]

[43] Jonathan Huggins, Ryan P Adams, and Tamara Broderick. 2017. Pass-glm: polynomial approximate sufficient statistics for scalable bayesian glm inference. *Advances in Neural Information Processing Systems* 30 (2017).

[44] Dimitrije Jankov, Shangyu Luo, Binhang Yuan, Zhuhua Cai, Jia Zou, Chris Jermaine, and Zekai J Gao. 2019. Declarative recursive computation on an rdbms, or, why you should use a database for distributed machine learning. *arXiv preprint arXiv:1904.11121* (2019).

[45] Dimitrije Jankov, Binhang Yuan, Shangyu Luo, and Chris Jermaine. 2021. Distributed numerical and machine learning computations via two-phase execution of aggregated join trees. *Proceedings of the VLDB Endowment* 14, 7 (2021).

[46] Manas Joglekar, Rohan Puttagunta, and Christopher Ré. 2015. Aggregations over generalized hypertree decompositions. *arXiv preprint arXiv:1508.07532* (2015).

[47] Manas R Joglekar, Rohan Puttagunta, and Christopher Ré. 2016. Ajar: Aggregations and joins over annotated relations. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. 91–106.

[48] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. 2017. Lightgbm: A highly efficient gradient boosting decision tree. *Advances in neural information processing systems* 30 (2017).

[49] Mahmoud Abo Khamis, Ryan R Curtin, Benjamin Moseley, Hung Q Ngo, XuanLong Nguyen, Dan Olteanu, and Maximilian Schleich. 2020. Functional Aggregate Queries with Additive Inequalities. *ACM Transactions on Database Systems (TODS)* 45, 4 (2020), 1–41.

[50] Mahmoud Abo Khamis, Hung Q Ngo, XuanLong Nguyen, Dan Olteanu, and Maximilian Schleich. 2018. AC/DC: In-database learning thunderstruck. In *Proceedings of the Second Workshop on Data Management for End-To-End Machine Learning*. 1–10.

[51] Tim Kraska, Ameet Talwalkar, John C Duchi, Rean Griffith, Michael J Franklin, and Michael I Jordan. 2013. MLbase: A Distributed Machine-learning System.. In *Cidr*, Vol. 1. 2–1.

[52] Arun Kumar, Jeffrey Naughton, and Jignesh M Patel. 2015. Learning generalized linear models over normalized data. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 1969–1984.

[53] Xupeng Li, Bin Cui, Yiru Chen, Wentao Wu, and Ce Zhang. 2017. Mlog: Towards declarative in-database machine learning. *Proceedings of the VLDB Endowment* 10, 12 (2017), 1933–1936.

[54] Mark Mucchetti. 2020. BigQuery ML. In *BigQuery for Data Warehousing*. Springer, 419–468.

[55] Kevin P Murphy. 2012. *Machine learning: a probabilistic perspective*. MIT press.

[56] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. 2015. Fast serializable multi-version concurrency control for main-memory database systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 677–689.

[57] Milos Nikolic and Dan Olteanu. 2018. Incremental view maintenance with triple lock factorization benefits. In *Proceedings of the 2018 International Conference on Management of Data*. 365–380.

[58] Dan Olteanu and Maximilian Schleich. 2016. Factorized databases. *ACM SIGMOD Record* 45, 2 (2016), 5–16.

[59] Dan Olteanu and Jakub Závodnỳ. 2015. Size bounds for factorised representations of query results. *ACM Transactions on Database Systems (TODS)* 40, 1 (2015), 1–44.

[60] The pandas development team. 2020. *pandas-dev/pandas: Pandas*. https://doi.org/10.5281/zenodo.3509134

[61] Johns Paul, Shengliang Lu, Bingsheng He, et al. 2021. Database Systems on GPUs. *Foundations and Trends® in Databases* 11, 1 (2021), 1–108.

[62] Judea Pearl. 1982. *Reverend Bayes on inference engines: A distributed hierarchical approach.* Cognitive Systems Laboratory, School of Engineering and Applied Science ….

[63] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[64] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an embeddable analytical database. In *Proceedings of the 2019 International Conference on Management of Data*. 1981–1984.

[65] Vijayshankar Raman, Gopi Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M Lohman, et al. 2013. DB2 with BLU acceleration: So much more than just a column store. *Proceedings of the VLDB Endowment* 6, 11 (2013), 1080–1091.

[66] Matthew Rocklin. 2015. Dask: Parallel computation with blocked algorithms and task scheduling. In *Proceedings of the 14th python in science conference*, Vol. 130. Citeseer, 136.

[67] Geetika Saxena and Bharat Bhushan Agarwal. 2014. Data Warehouse Designing: Dimensional Modelling and ER Modelling. *International Journal of Engineering Inventions* 3, 9 (2014), 28–34.

[68] Maximilian Schleich, Dan Olteanu, Mahmoud Abo Khamis, Hung Q Ngo, and XuanLong Nguyen. 2019. A layered aggregate engine for analytics workloads. In *Proceedings of the 2019 International Conference on Management of Data*. 1642–1659.

[69] Maximilian Schleich, Dan Olteanu, and Radu Ciucanu. 2016. Learning linear regression models over factorized joins. In *Proceedings of the 2016 International Conference on Management of Data*. 3–18.

[70] Ali Mohammadi Shanghooshabad, Meghdad Kurmanji, Qingzhi Ma, Michael Shekelyan, Mehrdad Almasi, and Peter Triantafillou. 2021. PGMJoins: Random Join Sampling with Graphical Models. In *Proceedings of the 2021 International Conference on Management of Data*. 1610–1622.

[71] Shreya Shankar, Rolando Garcia, Joseph M Hellerstein, and Aditya G Parameswaran. 2022. Operationalizing Machine Learning: An Interview Study. *arXiv preprint arXiv:2209.09125* (2022).

[72] Haijian Shi. 2007. *Best-first decision tree learning.* Ph.D. Dissertation. The University of Waikato.

[73] John B Smelcer. 1995. User errors in database query composition. *International Journal of Human-Computer Studies* 42, 4 (1995), 353–381.

[74] Mike Stonebraker, Daniel J Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, et al. 2018. C-store: a column-oriented DBMS. In *Making Databases Work: the Pragmatic Wisdom of Michael Stonebraker*. 491–518.

[75] Michael Stonebraker, Paul Brown, Alex Poliakov, and Suchi Raman. 2011. The architecture of SciDB. In *Scientific and Statistical Database Management: 23rd International Conference, SSDBM 2011, Portland, OR, USA, July 20-22, 2011. Proceedings 23*. Springer, 1–16.

[76] Swarup Acharya Phillip B Gibbons Viswanath and Poosala Sridhar Ramaswamy. 1998. Join Synopses for Approximate Query Answering. (1998).

[77] Reynold S Xin, Josh Rosen, Matei Zaharia, Michael J Franklin, Scott Shenker, and Ion Stoica. 2013. Shark: SQL and rich analytics at scale. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of data*. 13–24.

[78] Keyu Yang, Yunjun Gao, Lei Liang, Bin Yao, Shiting Wen, and Gang Chen. 2020. Towards factorized svm with gaussian kernels over normalized data. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 1453–1464.

[79] Zhuoyue Zhao, Robert Christensen, Feifei Li, Xiao Hu, and Ke Yi. 2018. Random sampling over joins revisited. In *Proceedings of the 2018 International Conference on Management of Data*. 1525–1539.

A DECISION TREE FROM SEMI-RING

In this section, we present the algorithms to express the criteria of reduction in variance for regression, information gain, gini impurity and chi-square for classifications based on the semi-ring in Table 1. Computing these criteria is at the heart of decision tree training. Our algorithm here is based on Sklearn[9].

Aggregated Semi-ring: It has been shown that [18, 68], for variance semi-ring, the aggregated semi-ring $\gamma(R_\bowtie)$ is a 3-tuple (C, S, Q) that represents the count $C = \sum_{t \in R_\bowtie} 1$, sum of target variable $S = \sum_{t \in R_\bowtie} t[Y]$, and the sum of squares of target variable $Q = \sum_{t \in R_\bowtie} t[Y]^2$. For classification with k classes, the aggregated semi-ring $\gamma(R_\bowtie)$ is a $(k+1)$-tuple $(C, C^1, ..., C^k)$ that represents the total count $C = \sum_{t \in R_\bowtie} 1$ and count of each class $C^i = \sum_{t \in R_\bowtie} \mathbf{1}_{t[Y]=i}$.

Reduction in variance for regression: In the scope of variance semi-ring and given $R_\bowtie = R_1 \bowtie R_2 \ldots \bowtie R_n$ and schema $S_1, \ldots, S_n$, let the average of $R_\bowtie[y]$ be $\hat{y}$. The total variance of the target variable in $R_\bowtie$ can be computed as

$$\begin{aligned}
var(R_\bowtie) &= \sum_{t \in R_\bowtie} (t[y] - \hat{y})^2 \\
&= \sum_{t \in R_\bowtie} t[y]^2 - 2 \sum_{t \in R_\bowtie} t[y] \cdot \hat{y} + \sum_{t \in R_\bowtie} \hat{y}^2 \\
&= \sum_{t \in R_\bowtie} t[y]^2 - \sum_{t \in R_\bowtie} \hat{y}^2 = Q - S^2/C
\end{aligned}$$

The third equation holds because $\sum_{t \in R_\bowtie} t[y] = \sum_{t \in R_\bowtie} \hat{y}$. Consider a selection predicate σ where the attributes in σ, which splits $R_\bowtie$ into $\sigma(R_\bowtie)$ and $\bar{\sigma}(R_\bowtie)$, and let the average of y in $\sigma(R_\bowtie)$ and $\bar{\sigma}(R_\bowtie)$ be $\hat{y}_\sigma$ and $\hat{y}_{\bar{\sigma}}$, respectively. Let their aggregated semi-ring be $\gamma(\sigma(R_\bowtie)) = (C_\sigma, S_\sigma, Q_\sigma)$ and $\gamma(\bar{\sigma}(R_\bowtie)) = (C - C_\sigma, S - S_\sigma, Q - Q_\sigma)$. Thus, the new variance of $\sigma(R_\bowtie)$ and $\bar{\sigma}(R_\bowtie)$ can be similarly computed as

$$\begin{aligned}
var(\sigma(R_\bowtie)) &= \sum_{t \in \sigma(R_\bowtie)} t[y]^2 - \sum_{t \in \sigma(R_\bowtie)} \hat{y}_\sigma^2 = Q_\sigma - S_\sigma^2/C_\sigma \\
var(\bar{\sigma}(R_\bowtie)) &= \sum_{t \in \bar{\sigma}(R_\bowtie)} t[y]^2 - \sum_{t \in \bar{\sigma}(R_\bowtie)} \hat{y}_{\bar{\sigma}}^2 \\
&= (Q - Q_\sigma) - (S - S_\sigma)^2/(C - C_\sigma)
\end{aligned}$$

Thus, the reduction in variance can be expressed as computations using variance semi-ring

$$\begin{aligned}
&var(R_\bowtie) - (var(\sigma(R_\bowtie)) + var(\bar{\sigma}(R_\bowtie))) \\
&= -S^2/C + S_\sigma^2/C_\sigma + (S - S_\sigma)^2/(C - C_\sigma)
\end{aligned}$$

As an optimization, we note Q term is canceled out for the reduction in variance. Therefore, we don't include Q during training.

To parse it into SQL for each feature A, we can compute its best split in one query using the window function. Note that, to avoid overflow, we need to compute s_t^2/c_t as $(s_t/c_t) \times s_t$.

```
SELECT A,-(stotal/ctotal)*stotal+(s/c)*s+(stotal-s)/(ctotal-c)*(stotal-s)
FROM (SELECT A, SUM(c) OVER(ORDER BY A) as c, SUM(s) OVER(ORDER BY A) as s
    FROM (  SELECT A, s, c
            FROM R
            GROUP BY A
        ))
ORDER BY criteria DESC
LIMIT 1;
```

---

[9]https://scikit-learn.org/stable/modules/tree.html#tree-mathematical-formulation

Classification: In the scope of class count semi-ring and given $R_\bowtie$, we will show that the criteria for classification (information gain, Gini impurity, Chi-square) can be computed given the aggregated class count semi-ring structures. Then it follows the same logic that we do not have to materialize the full $R_\bowtie$. Classification criteria are based on the probability of each class, which could be computed as $p^k = C^k/C$. The different criteria can be computed as:

$$entropy(R_\bowtie) = -\sum_{k=1}^{K} p^k \log p^k = -\sum_{k=1}^{K} (C^k/C) log(C^k/C)$$

$$gini(R_\bowtie) = 1 - \sum_{i=1}^{K} (p^k)^2 = 1 - \sum_{i=1}^{K} (C^k/C)^2$$

Consider a selection predicate σ where the attributes in σ, which splits $R_\bowtie$ into $\sigma(R_\bowtie)$ and $\bar{\sigma}(R_\bowtie)$. Let their aggregated semi-ring be $\gamma(\sigma(R_\bowtie)) = (C_\sigma, C_\sigma^1, ..., C_\sigma^k)$ and $\gamma(\bar{\sigma}(R_\bowtie)) = (C_{\bar{\sigma}}, C_{\bar{\sigma}}^1, ..., C_{\bar{\sigma}}^k) = (C - C_\sigma, C - C_\sigma^1, ..., C - C_\sigma^k)$. We can compute the reduction after the split in a way similar to regression.

Chi-square of the split is computed as:

$$\chi^2 = \sum_{i=1}^{K} \left(\frac{(C_\sigma^i - C^i C_\sigma/C)^2}{C^i C_\sigma/C} + \frac{(C_{\bar{\sigma}}^i - C^i C_{\bar{\sigma}}/C)^2}{C^i C_{\bar{\sigma}}/C} \right)$$

B BOOSTED TREES FROM SEMI-RING

In this section, we discuss the details of building Gradient Boosting from Semi-ring. We use the semi-rings as defined in Table 2.

B.1 Semi-ring Extension

Extension to bag semantics and weighted relations. Tuples in relations could be weighted. For relation R_i, let w be the function that maps tuple $t \in R_i$ to the weight as a real number. We modify the definition of semi-ring such that their count is a real number, then annotate relations as before, but their annotations are further multiplied by $(w(t), 0, 0)$ for weighting.

Extension to theta joins and outer joins. The support for theta joins over annotated relations is straightforward:

$$(R \bowtie_\theta T)(t) = \begin{cases} R(\pi_{S_R}(t)) \times T(\pi_{S_T}(t)), \theta(t) \\ Zero\ Element, \neg\theta(t) \end{cases}$$

To support outer-join, note that the semantic of annotated relation is that tuples not in the relation are annotated with zero-element [18]. This is undesirable as zero-element annihilates other elements when multiplied. Therefore for outer-join, we define the non-existed tuples to be annotated with one-element. Let $J = S_R \cap S_T$ be the join key between R and S:

$$(R\bowtie T)(t) = \begin{cases} R(\pi_{S_R}(t)) \times T(\pi_{S_T}(t)), \pi_{S_T}(t) \in T \\ R(\pi_{S_R}(t)), \pi_J(t) \notin \pi_J(T) \wedge \pi_{S_T - J}(t)\ ALL\ NULL \end{cases}$$

B.2 Boosted Trees Preliminary

We provide the background of Boosted Tree based on [28, 48, 55].

Objective. We consider ML model f that maps tuple $t \in R_\bowtie$ to the prediction. The prediction is a real number for regression, and a set of real numbers corresponding to the probabilities of classes

Semi-ring	Zero/One	Operator	Lift
Regression $(\mathbf{R}, \mathbf{R})$	**0:** $(0, 0)$ **1:** $(1, 0)$	$(h_1, g_1) + (h_2, g_2) = (h_1 + h_2, g_1 + g_2)$ $(h_1, g_1) \times (h_2, g_2) = (h_1 h_2, g_1 h_2 + g_2 h_1)$	$(h(t), g(t))$
Classification $((\mathbf{R}, \mathbf{R}), ..., (\mathbf{R}, \mathbf{R}))$	**0:** $((0,0), ..., (0,0))$ **1:** $((1,0), ..., (1,0))$	$((h_1^1, g_1^1), ..., (h_1^k, g_1^k)) + ((h_2^1, g_2^1), ..., (h_2^k, g_2^k))$ $= ((h_1^1 + h_2^1, g_1^1 + g_2^1), ..., (h_1^k + h_2^k, g_1^k + g_2^k))$ $((h_1^1, g_1^1), ..., (h_1^k, g_1^k)) \times ((h_2^1, g_2^1), ..., (h_2^k, g_2^k))$ $= ((h_1^1 h_2^1, g_1^1 h_2^1 + g_2^1 h_1^1), ..., (h_1^k h_2^k, g_1^k h_2^k + g_2^k h_1^k))$	$((h^1(t), g^1(t)), ..., (h^k(t), g^k(t)))$

<p align="center">Table 2: Gradient Semi-rings for gradient boosting.</p>

Task	Loss function	Gradient $g(\cdot)$	Hessian $h(\cdot)$	Prediction P
Regression	L2/rmse: $(\epsilon)^2$	ϵ	1	$mean(\mathcal{E})$
	L1/mae: $|\epsilon|$	$sign(\epsilon)$	1	$median(\mathcal{E})$
	Huber Loss: $\begin{cases} 0.5\epsilon^2, |\epsilon| \leq \delta \\ \delta(|\epsilon| - 0.5\delta), else \end{cases}$	$\begin{cases} \epsilon, |\epsilon| \leq \delta \\ \delta \cdot sign(\epsilon), else \end{cases}$	1	p^*
	Fair Loss: $c|\epsilon| - c^2 log(|\epsilon|/c + 1)$	$c\epsilon/(|\epsilon| + c)$	$c^2/(|\epsilon| + c)^2$	p^*
	Poisson Loss: $e^P - yp$	$e^P - y$	e^P	p^*
	Quantile Loss: $\begin{cases} (\alpha - 1)\epsilon, \epsilon < 0 \\ \alpha\epsilon, \epsilon \geq 0 \end{cases}$	$\begin{cases} \alpha - 1, \epsilon < 0 \\ \alpha, \epsilon \geq 0 \end{cases}$	1	$pctl_\alpha(\mathcal{E})$
	mape: $|y - p|/max(1, |y|)$	$\frac{sign(y-p)}{max(1,|y|)}$	1	$median(\mathcal{E})$
	Gamma Loss: $\frac{\frac{y}{p} - log(\frac{y}{p\psi})}{\psi} + log(y)$	$1 - ye^{-P}$	ye^{-P}	p^*
	Tweedie Loss: $-\frac{ye^{(1-\rho)log(p)}}{1-\rho} + \frac{e^{(2-\rho)log(p)}}{2-\rho}$	$-ye^{(1-\rho)P} + e^{(2-\rho)P}$	$-(1-\rho)ye^{(1-\rho)P} + (2-\rho)e^{(2-\rho)P}$	p^*
Classification	Softmax: $-y^k log(p^k)$	$p^k - y^k$	$\frac{K}{K-1} p^k(1 - p^k)$	p^*

Table 3: Summary of Gradient and Hessian implemented in the source codes from `LightGBM`. We note that Gradients and Hessians are not mathematically rigorous; they have been normalized and approximated for practical concerns. The residual $\epsilon = y - p$. $\mathcal{E}$ is the residual column. $pctl_\alpha$ is α percentile function.

for classification. We want to train a model f that minimizes the following objective:

$$L(f) = \sum_{t \in R_{\bowtie}} l(f(t), t[Y]) + \Omega(f)$$

Here, l is the loss function and Ω is the regularization function.

Regularization. For boosted tree of K iterations, f uses K additive functions $f_1, ..., f_K$ for predictions:

$$f(t) = \sum_{k=1}^{K} f_k(t)$$

The regularization term prefers tree models with smaller number of leaves and variance. For each tree f_i, let T_i be its number of leaves and p_i be a vector of leaf predictions. Then:

$$\Omega(f) = \alpha \sum_{k=1}^{K} T_k + 0.5\beta \sum_{k=1}^{K} ||p_i||^2$$

Optimization. Boosted Trees iteratively train decision trees to optimize the objective based on the predictions from previous trees. For ith tree f_k, it is optimizing:

$$L^i(f_i) = \sum_{t \in R_{\bowtie}} l(\sum_{k=1}^{i-1} f_k(t) + f_i(t), t[Y]) + \Omega(f_i)$$

After applying second optimization and removing constant terms:

$$\tilde{L}^i(f_i) = \sum_{t \in R_{\bowtie}} (g_t f_i(t) + 0.5 h_t f_i^2(t)) + \Omega(f_i)$$

where g_t and h_t are the gradients and hessians on the loss $l(\sum_{k=1}^{i-1} f_k(t), t[Y])$ for tuple t with respect to the predictions from previous trees $\sum_{k=1}^{i-1} f_k(t)$.

Next, we study the optimal prediction for leaf node. Consider a leaf node of decision tree f_i with number of leaves T_i, selection predicate σ and prediction p as a variable. The optimization objective (after removing constant terms) for this leaf node is then:

$$\tilde{L}^i_\sigma(p) = \sum_{t \in \sigma(R_{\bowtie})} (g_t p + 0.5 h_t p^2) + 0.5\beta p^2$$

The optimal prediction is then:

$$p^* = -\frac{\sum_{t \in \sigma(R_{\bowtie})} g_t}{\sum_{t \in \sigma(R_{\bowtie})} h_t + \beta}$$

And the objective becomes:

$$\tilde{L}_\sigma^i(p^*) = -0.5\frac{(\sum_{t\in\sigma(R_\bowtie)} g_t)^2}{\sum_{t\in\sigma(R_\bowtie)} h_t + \beta}$$

Finally, we study the problem of evaluating a split. Consider the selection predicate σ that splits $R_\bowtie$ into $\sigma(R_\bowtie)$ and $\bar{\sigma}(R_\bowtie)$. The initial loss before split is:

$$\tilde{L} = -0.5\frac{(\sum_{t\in R_\bowtie} g_t)^2}{\sum_{t\in R_\bowtie} h_t + \beta}$$

The loss after split is:

$$\tilde{L}_\sigma + \tilde{L}_{\bar{\sigma}} + \alpha = -0.5\big[\frac{(\sum_{t\in\sigma(R_\bowtie)} g_t)^2}{\sum_{t\in\sigma(R_\bowtie)} h_t + \beta} + \frac{(\sum_{t\in\bar{\sigma}(R_\bowtie)} g_t)^2}{\sum_{t\in\bar{\sigma}(R_\bowtie)} h_t + \beta}\big] + \alpha$$

The α term is because spliting increases the number of leaves. Finally, the reduction of loss from the split is:

$$\tilde{L} - (\tilde{L}_\sigma + \tilde{L}_{\bar{\sigma}} + \alpha)$$

The training process of Boosted Trees evaluates the reduction of loss for all candidate splits and chooses the best one.

B.3 Boosted Trees from Semi-ring

From the last section, we find that the core statistics over relational tables are the sum of gradients and hessians on the loss. Therefore, we build gradient semi-ring and lift relation based on the gradient and hessian for different objective functions, thus avoiding the materialization of $R_\bowtie$.

The definition of gradient semi-ring is in Table 2 and the formula of gradients and hessians for different objective functions based on LightGBM are in Table 3. We note that the gradients and hessians are not mathematical rigourious, and some losses are approximated. For example, mean absolute error has hessian undefined when error = 0, and 0 for the rest. Hessian can't be zero, and thus the loss is approximated without second order gradient (so hessian all 1) [10].

C OTHER EXPERIMENTS

C.1 Comparison with LightGBM

Figure 17 shows the experiment results for TPC-DS with SF=10. JoinBoost is $\sim 3\times$ faster than LightGBM for random forests, and $\sim 1.3\times$ faster than LightGBM for gradient boosting. For TPC-H, JoinBoost is slowed because of the large dimension tables (Orders and PartSupp). The messages between the fact table and these two dimension tables are large but expensive. To address this, the junction hypertree could be redesigned to reduce message sizes [42] as future works.

C.2 Parallelization

We use JoinBoost with (para) and without (w/o) inter-query parallelism on Favorita, but always with intra-query optimization (4 threads per query for para and 16 threads per query for w/o), to train a tree of 8 leaves. Figure 18 shows the result. Gradient Boosting and Random Forest with 100 trees Figure 18 shows the model training time for 100 trees. For gradient boosting, the training time is reduced by 28% by exploiting partial dependency in queries. For
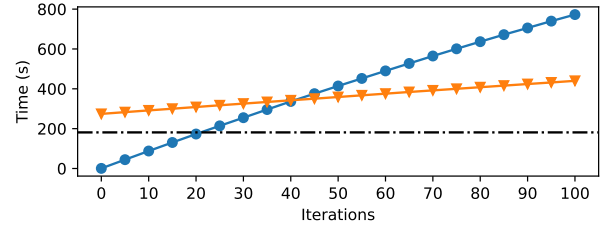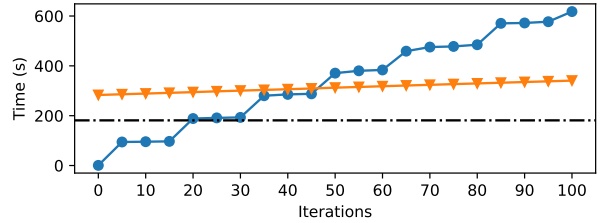
---

[10]https://github.com/microsoft/LightGBM/pull/175
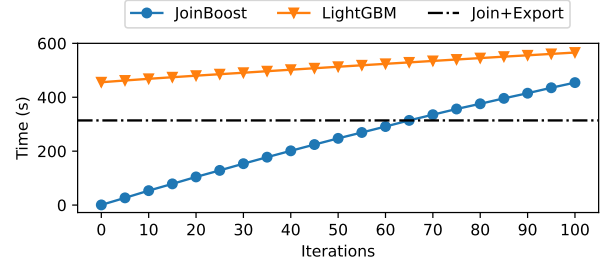
(a) Gradient Boosting for TPC-DS.
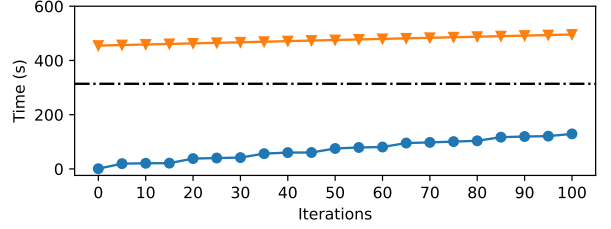
(b) Random Forest for TPC-DS.

(c) Gradient Boosting for TPC-H.

(d) Random Forest for TPC-H.

Figure 17: Gradient Boosting and Random Forest training time and model performance compared to the SOTA ML Frameworks. Dotted black line is the overhead of materializing the join and export join by DuckDB.

Random Forest, the tree-wise parallelism reduces the training time more significantly by 35%.

D OTHER OPTIMIZATIONS

D.0.1 Data Transformations. Prior factorized learning work [50] introduced techniques that use functional dependencies to prune attributes prior to training, and to avoid materializing one-hot encodings for categorical attributes. A benefit of a middle-ware
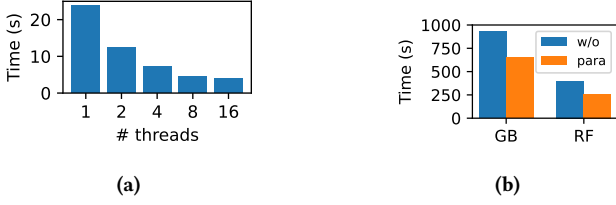
(a)   **(b)**

Figure 18: (a) Training time of 1 tree of 8 leaves for intra-query parallelism with a varying number of threads. (b) Training time of Gradient Boosting and Random Forest with (para) and without (w/o) inter-query parallelism.
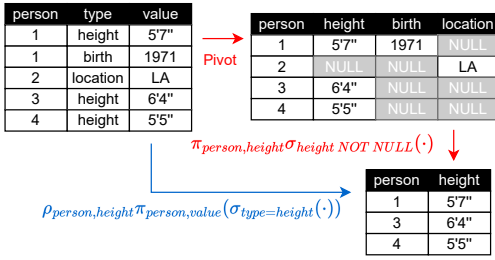

Figure 19: Pivot Transformation Optimizations over *Person_Info* **in IMDB. Naively, the sparse pivot result is materialized, then each feature (e.g., height) is selected to evaluate the split criteria (red). The query could be rewritten as a selection over type column by feature name, thus avoiding the materialization of a sparse pivot table (blue).**

approach to in-DB ML is that we can leverage existing query optimizations to support other data transformations. In addition to the prior techniques above, `JoinBoost` introduces an optimization to support "pivot" transformations.

ML algorithms traditionally expect a dense matrix representation of the features. In contrast, DBMSes store sparse representations that need to be pivoted for ML libraries. Figure 19 illustrates IMDB data containing attribute-value pairs (e.g., birth, 1971) that encode information about a person.

If `type` were used as a feature, we first need to pivot the attribute into a sparse table with one column for each `type` value (e.g., height, birth). This matrix is then aggregated to compute statistics during training. To avoid this, Cunningham et al. [31] describe an optimization that rewrites aggregations over pivots as selections over the original table. `JoinBoost` easily supports such optimizations simply by adding an additional rewriting step. For the IMDB dataset, this optimization speeds up node splits by 3.8× for the *Person_Info* table, compared to naive pivoting.

D.0.2 Missing Join Keys. ML libraries like `LightGBM` and `XGBoost` provide native support for missing values because they are ubiquitous in real-world datasets. They evaluate losses by allocating missing values to either side of a candidate split and choosing the best one. `JoinBoost` efficiently supports this: candidate splits are already computed using a group-by, so the missing values will be aggregated in a group. We simply add the missing value's annotation to each split candidate and choose the best. The overhead is negligible compared to the aggregation itself.
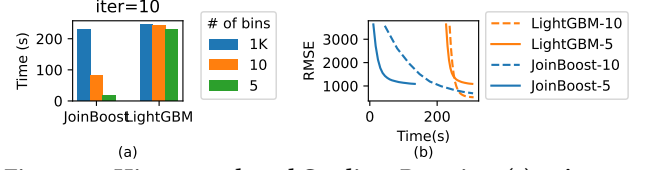

(a)   **(b)**

Figure 20: Histogram-based Gradient Boosting. (a) `JoinBoost` is much faster with fewer bins due to the smaller cuboid sizes. (b) Training time-accuracy trade-off. `JoinBoost` with #bin = 5 pushes the Pareto frontier and reaches *RMSE* = 1500 in 20s.

What if the join key is missing values, as this affects the join's output cardinality? Semi-ring annotations naturally support this: message passing simply uses full outer joins instead of inner joins, and then applies the procedure above.

Identity and Semi-join Message Optimizations. Snowflake schemas often exhibit a property that allows us to drop messages (and skip the associated joins) along *Identity Paths* during message passing. An identity path starts from a leaf relation (usually a dimension table), and flows along 1-to-N relationship edges, as long as each relation is not R_Y and no join key is missing. The key property is that semi-ring addition (aggregation) does not apply along the identity path due to the 1-to-N join relationships.

Let $L \rightarrow P$ connects leaf relation L to P. Each tuple $\in L$ is annotated with the 1 semi-ring element, and each tuple in P joins with exactly one tuple in L. $m_{L \rightarrow P}$ is an *Identity Message* because it doesn't change P's annotations, and thus can be dropped. Relations along the identity path are all initially annotated with 1, so this applies recursively. This optimization is used when splitting the root decision tree node.

Now suppose the best split was on attribute l in L, so the edge is now $\sigma_l(L) \rightarrow P$. Although L does not emit an identity message, we know that each tuple in the message is annotated with 1. Thus, we don't materialize the semi-ring elements as columns and can rewrite the join with P as a semi-join, and simply filter P by the join values in $\sigma_l(L)$.

D.0.3 Histogram-based Cuboid. A popular technique implemented in `LightGBM` and `XGBoost` is histogram-based gradient boosting. It computes histograms over each feature, and replaces each feature's value with its bin id (the cardinality is the same). Varying the number of bins trades off accuracy for lower training time.

A natural optimization is to leverage data cubes [35] when the number of bins is small and the data is sparse. We evaluate a naive approach that materializes the full dimensional cuboid—`GROUP BY` all feature attributes—and uses it instead of factorized learning to execute the aggregation queries for training. On the Favorita dataset, 5 bins result in a cuboid with $3M$ rows, as compared to 80M for the full join result. Since training is based on semi-ring annotations, the rest of the algorithms remain the same.

For the experiment, we vary the number of histogram bins to study when using the cuboid optimization (Appendix D.0.3) is effective. For bin=5, 10, 1K, the cuboid contains ~3M, ~25M, and ~80M rows respectively.

Figure 20(a) shows that at iteration 10 and $bin = 5$, `JoinBoost` speeds up by >100× the smaller cuboid size and faster aggregation queries, whereas `LightGBM` only slightly improves. As we run more boosting iterations (Figure 20(b)) `JoinBoost-5` quickly converges

($\sim 20s$); `JoinBoost-10` takes longer to converge because the cuboid is larger but is still competitive to `LightGBM`. In theory, we could gradually increase the number of bins throughout training to further improve model accuracy, but leave this to future work.