JoinBoost: In-Database Tree-Models In Action

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

Tree-models remain dominant on tabular datasets, which are typically normalized and stored in DBMSes. Specialized ML libraries (e.g., `LightGBM`, `XGBoost`) require developers to materialize and export a single denormalized table from the DBMS, import it into ML library, and then train the model; this join, materialization and export process is expensive and poses security risks. In-DB ML systems offer a solution by training ML within DBMSes, providing strong data governance. Ideally, they should be portable to any DBMS (e.g., by using only SQL) and also offer competitive performance to specialized ML libraries. However, DBMSes are notoriously slow for ML, and current In-DB ML systems rely on specialized optimizations that are not portable. Additionally, they are not optimized for the normalized schemas common in DBMSes.

We present `JoinBoost`, the first In-DB ML system for tree-models that achieves both portability and performance. To optimize for normalized schemas, `JoinBoost` extends factorized ML to tree-models with optimized algorithms. Unlike previous factorized ML that relies on specialized engines, `JoinBoost` is highly portable: `JoinBoost` uses a novel architecture that issues SPJA queries to DBMSes for ML and applies optimizations through pure query rewriting. Our experiment shows that `JoinBoost` is over an order magnitude faster than SOTA In-DB and factorized ML systems; compared to `LightGBM`, `JoinBoost` is within 2× for gradient boosting and 3× faster for random forests. However, `JoinBoost` scales well beyond `LightGBM` in terms of the number of features, database size, and join graph complexity (snowflake and galaxy schemas).

PVLDB Reference Format:
Zezhou Huang, Rathijit Sen, Jiaxiang Liu, and Eugene Wu. `JoinBoost`: In-Database Tree-Models In Action. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

PVLDB Availability Tag:
The source code of this research paper has been made publicly available at http://vldb.org/pvldb/format_vol14.html.

1 INTRODUCTION

Tree-based models—ranging from decision trees, random forests, and gradient boosting—are recursive space-partitioning algorithms used for classification and regression [53]. In contrast to the popular neural network models, tree models remain the most effective [34] and popular models for tabular datasets. For instance, random forests and gradient boosting were rated the most popular ML model from 2019 to 2021 on Kaggle [7]. In response, the ML community has developed optimized tree-training libraries, such as `LightGBM` [44] and `XGBoost` [22], that offer easy-to-use API, excellent single-node performance, and can run on distributed frameworks like Spark [75] or Dask [67].

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, the 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 4) 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 compute 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 system, and benefit from the excellent security, performance, and scalability features in modern DBMSes. Unfortunately, existing approaches like `MADLib` [35] are much slower than ML libraries, specific to a single DBMS design (e.g., PostgreSQL), and still require join materialization.

Factorized ML [45, 46, 57, 68] avoids join materialization by treating model training as semi-ring aggregations [33] over the join graph, and pushing the aggregations through the joins. The recent `LMFAO` [68] applies factorized ML to decision trees. Since evaluating a node split amounts to executing a batch of group-by aggregations (over the join graph) for each candidate feature, `LMFAO` uses work-sharing to optimize the entire batch of aggregations. Unfortunately, it is a custom engine external to the user's DBMS, incurs optimization and compilation overhead for every node split, and is in practice, limited to a single root node.

We believe a practical In-DB solution should meet a set of ambitious criteria. It should (C1) offer training performance competitive with the SOTA ML libraries (e.g., `LightGBM`), and also scale to massive data warehouses; (C2) be easily portable to any DBMS without the need for custom UDFs nor database extensions. However, these two criteria are often in tension: Using only the SQL interface to achieve portability can lead to slow performance, as databases

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

are not optimized for ML tasks [75]. On the other hand, prior approaches that aim for competitive performance rely on specialized optimizations such as external linear algebra libraries [28, 35] or GPU acceleration [15, 36, 60], which are not portable.

To this end, we developed JoinBoost[1], the first In-DB ML system for tree-models that achieves both criteria. JoinBoost is a lightweight Python library that supports decision trees, random forests, and gradient boosting. JoinBoost mirrors LightGBM's training API, trains and returns identical models to LightGBM, but translates the training algorithms over a join graph into simple Select-Project-Join-Aggregation (SPJA) queries that can be run on any DBMS or data frame library[2]. **Compared to LMFAO and MADLib, JoinBoost is over** 10× **and** 16× **faster. In small databases, JoinBoost is within** 2× **to train gradient boosted trees, and** 3× **faster to train random forests compared to LightGBM. However, JoinBoost scales in the # of features, join graph complexity, and dataset size (TPC-DS SF=1000 in Section 6.4) well beyond where LightGBM runs out of memory.**

JoinBoost makes a number of systems and algorithmic contributions. First and foremost, it proposes a novel architecture that allows DBMSes to support tree-models with minimal or no modification, and achieve performance comparable to specialized ML systems (e.g., LightGBM). This is made possible through the use of pure query rewriting to translate factorized ML into SQL queries, rather than relying on specialized optimizations as previous in-DB or factorized ML systems have done [15, 28, 35, 36, 60, 68, 69]. This enables JoinBoost to train random forests 3× faster.

Second, we identified and optimizes column-wise updates, the bottleneck for gradient boosting in current DBMSes. To compete with LightGBM, update performance needs to be similar to a parallelized write to an in-memory byte array. This can be natively optimized by columnar DBMSes using copy-on-write semantics. We simulated this by combining DuckDB [65] (for aggregation and join) with Pandas (for copy-on-write column updates), resulting in gradient boosting performance within a factor of 2×.

Third, JoinBoost extends factorized ML algorithms from batch optimization of decision stump [68] to support iteratively training a full tree over a join graph. A key observation is that intermediate results can be cached and reused across node splits (batches in [68]), which improves decision tree and random forest training by 3×.

Forth, and our key algorithm innovation, is to extend factorized ML to support boosting. Each boosting iteration trains a decision tree on the *residuals* from the previous tree, so it needs to update the target Y attribute to the residual in the denormalized table *that is not materialized*. However, this form of view update is ambiguous [23]. To address this, we logically rewrite residual updates over $R_{\bowtie}$ into a join between $R_{\bowtie}$ and an *update relation* U; U contains an attribute for the residual, along with all attributed reference in the decision tree. We are able to do this by leveraging the unique characteristics of tree-based models—that the leaves form a non-overlapping partition of the training data.

Finally, JoinBoost ports many pragmatic features from LightGBM, such as common data transformations (one-hot encoding, dictionary encoding), histogram-based gradient boosting, and support for missing-values. Beyond porting existing features, JoinBoost also introduces optimized pivoting, data cubes for histogram-based gradient boosting, and support missing values in join keys.

Note: The paper is self-contained and references to appendices can be skipped, or can be found in the technical report [9].

[1]Open-sourced at https://anonymous.4open.science/r/JoinBoost-FBC4

[2]Currently supports DuckDB, MariaDB, MonetDB, MySQL, Oracle, PostgreSQL, SparkSQL, SQLite, SQLServer, Redshift, Azure Synapse Analytics, BigQuery, and Pandas.

2 RELATED WORK

Tree-based ML Libraries. Random forest [18] and gradient boosting [30] are the de-facto ensemble models supported by almost all standard ML libraries including Sklearn [62], TensorFlow [11] and Keras [24]. The state-of-the-art Tree-based ML libraries are LightGBM [44] and XGBoost [22]. Both specialize in only Tree-based models, are heavily optimized and outperform the rest according to previous benchmarks [14]. According to Kaggle 2021 surveys [7], 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.

Tree-based models are well-studied, backed by theoretic guarantees [51, 63], and have a large number of variants. To better handle categorical attributes, CatBoost [64] proposes novel permutation-based statistics to encode categorical attributes. To support uncertainty estimation, NGBoost [27] proposes gradient boosting algorithms for probabilistic predictions. For tree growth, extra trees [31] and oblivious trees [47] could be applied to avoid overfitting. Gradient boosting has also been studied under online setting [16] with theoretical guarantees. These works are complimentary to this paper and JoinBoost could be extended to support these algorithms.

Factorized ML systems. There are two classes of factorized ML. The first class decomposes ML models into aggregation queries over semi-ring and pushes the marginalization before the join. They support popular ML models like ridge regression, decision stump [68], support vector machine [45] and factorization machine [68]. Other models like k-means [26], and Generalized Linear Model [38] can be approximated by semi-ring. Compared to traditional ML algorithms, factorized algorithms achieve asymptotically lower time complexity. There have been many research prototypes of factorized machine learning systems [26, 46, 68, 69] with specialized query execution engines implemented in C++ and show great performance improvement.

However, previous systems are not in-DB ML: these systems build query execution engine *outside of DBMSes* from scratch, thus require data export from DBMSes and demand large engineering efforts to maintain. Currently, these systems are used by experts but not widely used by the general database or machine learning audiences. Tree-based models are not supported by most of them, and LMFAO [68] supports them in a limited way (see Section 6.1). We extend the algorithms from this class of factorized ML and develop practical factorized tree-model ML system inside DBMSes.

The second class of factorized ML works focuses on the operator-level optimization of ML over joins [21, 48, 76] by carefully iterating over multiple tables, caching the intermediate results, and combining these results. Compared to the first class of factorized ML, these works support a broader range of ML algorithms like k-means and Generalized Linear Models (without approximation). However,

these works only reduce the space complexity not the time complexity, and can perform worse than naive join when the data fit into memory [48].

In-databases ML systems. Enterprise data sit within data warehouses, so running ML within the DBMS is a natural demand. To satisfy demands, cloud database vendors like Azure [1], Redshifts [4] and Snowflake [3] support ML; they however are not in-databases as they move data outside of databases to external compute clusters for ML, which is costly and insecure. To address these issues, `MADLib` [28, 35] extends PostgreSQL with statistical methods over matrices (e.g., linear-algebra). It uses a Python UDF driver to partition the matrices and invoke efficient linear algebra routines. These linear algebra routines are unnecessary for tree-based models that only need simple aggregations. Like `JoinBoost`, BigQuery [52] implements ML using pure SQL, but is limited to Generalized Linear Models and doesn't apply factorized ML.

Many works run ML on distributed [17, 40, 41, 75] or GPU [15, 36, 60] DBMSes, but don't support factorized ML. `JoinBoost` supports distributed, GPU, and cloud DBMSes, but the focus of this paper is in the single-node CPU setting. `JoinBoost` currently supports cloud DBMSes (Section 6.4), but we leave optimizations (e.g., data and model distribution) for them to future works.

3 BACKGROUND

In this section, we provide the background of factorized query execution, and 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 [13, 58].

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. Throughout examples, we also include the schema in the square bracket followed by the relation $R[A_1, \cdots, A_n]$ to help understanding. The domain of R is then the Cartesian product of attribute domains $dom(R) = dom(A_1) \times \cdots \times dom(A_n)$.

Annotated Relations. The annotated relational model [33, 42, 56] maps each $t \in R$ to a commutative semi-ring $(D, +, \times, 0, 1)$, where D is a set, $+$ and $\times$ are commutative binary operators closed over D, and $0/1$ are the zero/unit elements. These annotations form the basis of provenance semi-rings [33], 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:

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

$$(R \bowtie T)(t) = R(\pi_{S_R}(t)) \times T(\pi_{S_T}(t)) \quad (2)$$

(1) The annotation for each group-by result in $\gamma_{\mathbf{A}} R$ sums the annotations of all tuples in its input group. (2) The annotation of each join result in $R \bowtie T$ is the product of semi-ring annotations from its contributing tuples in R and T.

Aggregation Pushdown. The key optimization in factorized query execution [13, 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 result 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* [61]. While *Message Passing* supports general SPJA queries [37], it is sufficient for tree models to restrict ourselves to SPJA queries with zero (γ) or one (γ_A) group-by attribute. Message passing works as follows:

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[3].

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 [19], and its extensions (e.g., bagging and boosting) follow standard ML textbooks [53].

[3]The algorithm applies to acyclic join graphs. Cyclic join graphs can be pre-joined into acyclic join graphs using standard hypertree decomposition [13, 43].

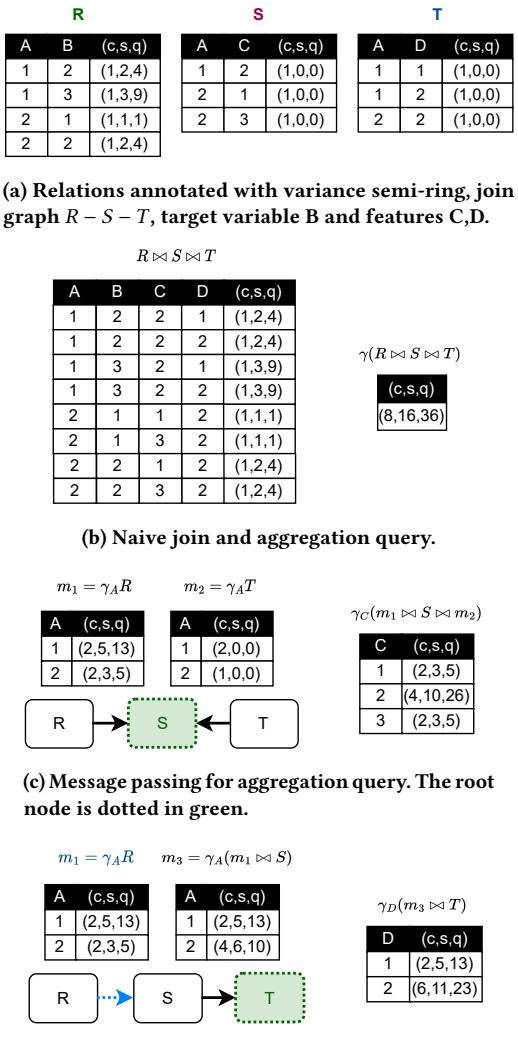
Semi-ring	Zero/One	Operator	Lift
Variance (Z, R, R)	**0:** $(0,0,0)$ **1:** $(1,0,0)$	$(c_1,s_1,q_1)+(c_2,s_2,q_2)=(c_1+c_2,s_1+s_2,q_1+q_2)$ $(c_1,s_1,q_1)\times(c_2,s_2,q_2)=(c_1c_2,s_1c_2+s_2c_1,q_1c_2+q_2c_1+2s_1s_2)$	$(1,y,y^2)$
Class Count (Z, Z, ..., Z)	**0:** $(0,...,0)$ **1:** $(1,...,0)$	$(c_1,c_1^1,...,c_1^k)+(c_2,c_2^1,...,c_2^k)=(c_1+c_2,c_1^1+c_2^1,...,c_1^k+c_2^k)$ $(c_1,c_1^1,...,c_1^k)\times(c_2,c_2^1,...,c_2^k)=(c_1c_2,c_1^1c_2+c_1c_2^1,...,c_1^kc_2+c_1c_2^k)$	$(1,0,...,c^y=1,...,0)$

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.

Decision Tree. Decision Tree is a function that maps (predicts) target variable Y from a set of features $\mathbf{X}$. Internally, decision tree stores a tree structure where each edge is associated with a predicate σ over single attribute $X \in \mathbf{X}$, and each leaf is associated with some prediction value $p \in dom(Y)$. Decision Tree makes prediction for $t \in Dom(\mathbf{X})$ by traversing the tree edges where t satisfies the edge predicate till the leaf and outputting the leaf prediction. We represent decision tree as a set of selection prediction pairs $\{(l.\sigma, l.p)\}$ for each tree leaf l, where $l.\sigma$ is the conjunction of predicates along the path from the root to l and $l.p$ is leaf prediction. The set of $l.\sigma$ in decision tree are mutually exclusive and collectively exhaustive in $Dom(\mathbf{X})$.

Given relation R with a set of features $\mathbf{X}$ and target variable Y where $\mathbf{X} \cup \{Y\} \subseteq S_R$, the training process of decision tree over R recursively splits R to minimize some criterion $c(\cdot)$. Particularly, the most popular criteria are *Variance* for regression, and *Gini Impurity, Entropy and Chi-Square* for classification. The splits for numerical attribute $A \in \mathbf{X}$ is inequality-based $(\sigma_{A>v}, \bar{\sigma}_{A\leq v})$, while the splits for categorical attribute A could be equality-based $(\sigma_{A=v}, \bar{\sigma}_{A\neq v})$ or set-based $(\sigma_{A\in V}, \bar{\sigma}_{A\notin V})$ using greedy algorithm [29]. 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 greedily with the largest criteria reduction. Finally, each leaf prediction is average Y for regression or mode Y for classification.

Bagging and Boosting. Large deep decision trees easily overfit, so ensemble techniques like bagging and boosting combine many weak learners (e.g., decision trees with smaller depth or leaves) to produce a more robust model. The most popular tree-based ensemble models are random forest [18] (bagging) and gradient boosting [30] (boosting). Random forest trains multiple decision trees over samples of R independently in parallel and aggregates their predictions (e.g., average) for the final prediction; Gradient boosting sequentially trains the next decision tree based on gradients of the criterion function with respect to the prediction from the previous decision trees.



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

(b) Naive join and aggregation query.

(c) Message passing for aggregation query. The root node is dotted in green.

(d) Message sharing between aggregation queries. The reusable message is dotted in blue.

Figure 1: Factorized decision stump training.

3.3 Factorized Learning for Tree-Stumps

We now introduce tree models over joins, and how factorized execution can accelerate their training. We consider a database with a set of relations $\mathcal{R} = \{R_1, R_2, ..., R_n\}$, we want to train a tree-based model 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). To keep the text simple, we will assume natural join, set semantics, and the Variance split criteria; Appendix B describes extensions to theta and outer joins, bag semantics, and formula for classification semi-rings.

Factorized learning avoids materializing $R_{\bowtie}$ by expressing learning algorithms as semi-ring aggregation queries, and applying aggregation pushdown over join. Let us first describe semi-ring annotations for trees and then the training algorithms.

Tree Semi-rings. We now illustrate how to use variance semi-ring (Table 1) to compute the Variance criterion for regression trees.

Each semi-ring defines a $lift(\cdot)$ function [56] that annotates a base tuple with its appropriate semi-ring element. Variance semi-ring initially annotates each tuple $t \in R_Y$ with $R_Y(t) = (1, t[Y], t[Y]^2)$, and tuples from the remaining relations with the 1 element $(1, 0, 0)$. During message passing, the annotations are combined via + and $\times$ as defined in Table 1. The aggregated semi-ring $\gamma(R_{\bowtie})$ is then a 3-tuple (C, S, Q) that represents the count, sum of Y, and the sum of squares of Y. The variance statistic is easily derivable from this aggregated semi-ring as $variance = Q - S^2/C$. In this way, any filter aggregation query over the join graph is expressible via message passing and lightweight post-processing.

Example 1. *Consider the database $\mathcal{R} = \{R, S, T\}$ in Figure 1a annotated 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 column B. We can compute $(C, S, Q) = \gamma(R \bowtie S \bowtie T) = (8, 16, 36)$ and $variance = Q - S^2/C = 36 - 16 \times 16/8 = 4$*

Factorized Decision Stump and Message Caching. Factorized learning relies on semi-ring aggregations to avoid materializing $R_{\bowtie}$. 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 query via message passing one by one is wasteful because messages could be reused across queries. Consider the example in Figure 1:

Example 2. *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 [37] shows that a simple caching scheme that materializes all messages between relations in the join graph (in both directions) is very effective in a range of analytical workloads. The factorized learning system LMFAO [68] also performs batch query optimization for splitting a single decision tree node. In the special case of decision trees, where each query groups by at most one feature, its optimizations are equivalent to this simple caching scheme. Algorithmically, LMFAO focuses on optimizing the batch of queries for a single decision tree node at a time and misses sharing opportunities across nodes. It also does not support residual updates necessary for boosting. In practice, it incurs considerable overhead (optimization and query compilation) for each query batch.

4 FACTORIZED TREE-BASED MODELS

This section discusses the algorithms of factorized tree-based models. The key novelties are (1) for the decision tree, we share the computations to evaluate the split criteria for not only expanding a single tree node, but also *across levels in the tree*; (2) we study the problem of updating residuals, which opens up a new class of boosting models like gradient boosting.

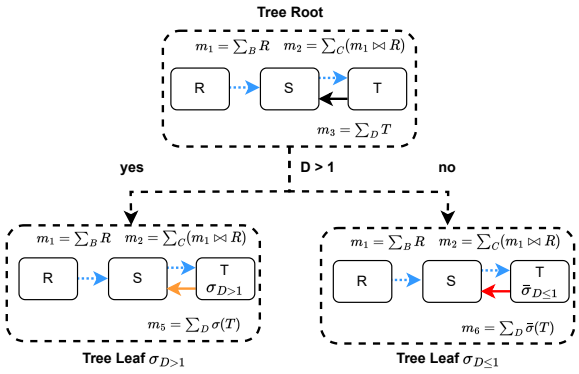


Figure 2: Share computations between the parent node and children nodes after the split. Dotted messages are shared.

4.1 Factorized Decision Tree

Previous factorized learning algorithms [46, 68] are based on batch optimization of aggregation queries, which is sufficient for models with closed-form solutions (e.g., ridge regression [68] or factorization machine [68]). However, the training process of tree-based models is iterative: the aggregation queries for the children nodes depend on the split (query results) from the parent node and can't batch ahead. As a result, previous algorithms have to either forgo the work-sharing opportunities between tree nodes, or batch the aggregation queries for all possible splits (which is impractical as the number of possible splits grows exponentially in tree depth).

Our key observation is that messages as intermediate results can not only be shared by the batch of aggregation queries within the tree node, but also among tree nodes. Consider the example:

Example 3. *Following Example 2, suppose the best split for the tree root is $\sigma_{D>1}$. The split applies $\sigma_{D>1}$ and $\bar{\sigma}_{D\leq 1}$ to relation T as it contains attribute D. After the split, there are two leaf nodes with relations: $\sigma(R_{\bowtie}) = R \bowtie S \bowtie \sigma(T)$ and $\bar{\sigma}(R_{\bowtie}) = R \bowtie S \bowtie \bar{\sigma}(T)$. Then, for each leaf node, we need to recursively compute the batch of aggregation queries and identify the next split through Message Passing. As illustrated in Figure 2, for both leaf nodes, all messages (in blue) along the path $R \rightarrow S \rightarrow T$ are the same as those from the tree root. We only need to recompute message $m_{T \rightarrow S}$ in each tree leaf. Note that the message $m_{S \rightarrow R}$ is skipped because R's attributes are not used as features in this model.*

In general, after a split over an attribute in relation R_i, all messages along the path from leaves to R_i can be re-used. The proof is as follows: all messages from the leaf relations (excluding R_i) are unchanged, and recursively, if a relation's (excluding R_i) incoming messages are unchanged, its outgoing message is also unchanged. This is orthogonal to prior batch optimization work [68] because we can cache and reuse the messages after batching for future nodes. This further improves batch optimization by >3× (Section 6.1).

4.2 Factorized Random Forest Models

A random forest model simply trains multiple decision trees over random samples of the training data, and aggregates (e.g., averages)

their predictions during inference. The main challenge is to efficiently draw a uniform sample over non-materialized $R_{\bowtie}$. We use ancestral sampling [20]: starting from a root relation in the join graph, we draw a sample s, filter the child relations by tuples that join with s, and recurse through the join graph. To ensure uniformity over $R_{\bowtie}$, the sampling is weighted. Given the current sample s, tuple t from the next relation R is weighted by $P(t|s) = P(t,s)/P(s)$. The probability $P(t)$ for $t \in R$ is proportional to the number of tuples $t_{\bowtie}$ in $R_{\bowtie}$ with $\pi_{S_R}(t_{\bowtie}) = t$ [70, 77]. This is efficiently computed with message passing and caching [37, 70].

Minor Optimizations. First, we coalesce the messages for *COUNT* query with those for 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 [74].

4.3 Factorized Boosted Tree-based Models

Boosted tree-based Models like gradient boosting [30] iteratively train the next tree to predict the residuals from the previous trees. In each iteration, we need to update the target variable to be the residual $\mathcal{E} = Y - P$, where P is the current prediction. In other words, for each leaf node l in the last decision tree, we logically want to update the subset of $R_{\bowtie}$ matching $l.\sigma$ to the difference between y and $l.p$. The fundamental challenge is: how to apply these updates without materializing $R_{\bowtie}$?

A tempting approach is to treat this as a view update problem: we wish to update the attributes in the view of $R_{\bowtie}$ and translate it into updates over the base relations. However, the general problem is well known to be non-deterministic [23]. A related approach is factorized IVM [39, 56], but that updates $R_{\bowtie}$ given changes to the base relations, which is the reverse of our problem.

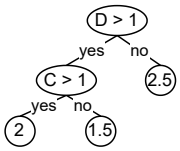
Update Relation U: While view updates are hard in general, residual updates are a special case because the UPDATEs are guaranteed to be non-overlapping, so we can rewrite the set of UPDATEs into a join with an *Update Relation* U. The benefit is that we can apply factorized learning over join.

To construct U, recall from Section 3.2 that a decision tree is modeled as a set of (predicate, prediction) pairs. In addition, each tuple in $R_{\bowtie}$ corresponds to exactly one leaf node. Let $\mathbf{A}$ be the set of attributes referenced by the predicate in the decision tree. U contains the projection of $\mathbf{A}$ over $R_{\bowtie}$, as well as its leaf's prediction.

The next boosting iteration requires computing $\mathcal{E} = Y - P$ over $R_{\bowtie} \bowtie U$, and lifting the resulting relation based on $\mathcal{E}$: $lift_{\mathcal{E}}(U \bowtie R_{\bowtie})$ in order to correctly annotate each tuple[4]. In general, $lift_{\mathcal{E}}$ cannot be pushed before the join, so $R_{\bowtie} \bowtie U$ would need to be materialized. The challenge is to *annotate* U with semi-ring annotations such that: $Annotate(U) \bowtie R_{\bowtie} = lift_{\mathcal{E}}(U \bowtie R_{\bowtie})$

Variance Semi-ring. The primary semi-ring for gradient boosting is the variance semi-ring, which is amenable to construction. We define $Annotate(U) = lift_{-P}(U)$—given a tuple $u \in U$ with prediction $u[P] = p$, its annotation is $(1, -p, p^2)$. Note that defining $Annotate(U)$ for an arbitrary semi-ring structure is non-trivial, and we leave this exploration to future work.

[4]Because we lift relations based on different target variable Y, we add the target variable as a subscript $lift_Y$ to disambiguate.

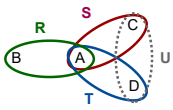


(a) Decision Tree.

U

C	D	P	(c,s,q)
1	1	2.5	(1,-2.5,6.25)
1	2	1.5	(1,-1.5,2.25)
2	1	2.5	(1,-2.5,6.25)
2	2	2	(1,-2,4)
3	2	2	(1,-2,4)

(b) Update Relation U.



(c) Join Graph.

A	B	C	D	P	$\mathcal{E}$	(c,s,q)
1	2	2	1	2.5	-0.5	(1,-0.5,0.25)
1	2	2	2	2	0	(1,0,0)
1	3	2	1	2.5	0.5	(1,0.5,0.25)
1	3	2	2	2	1	(1,1,1)
2	1	1	2	1.5	-0.5	(1,-0.5,0.25)
2	1	3	2	2	-1	(1,-1,1)
2	2	1	2	1.5	0.5	(1,0.5,0.25)
2	2	3	2	2	0	(1,0,0)

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

Figure 3: 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.

Proposition 4.1. *For any tuple* $t \in (U \bowtie R_{\bowtie})$, $lift_{\mathcal{E}}(U \bowtie R_{\bowtie})(t) = (lift_{-P}(U) \bowtie R_{\bowtie})(t)$.

Proof. Let $y = t[Y]$ and $p = t[P]$. Consider the materialized result of $U \bowtie R_{\bowtie}$. It contains target variable Y and prediction P, so t's residual is $t[\mathcal{E}] = y - p$. Thus, $lift_{\mathcal{E}}(U \bowtie R_{\bowtie})(t) = (1, y - p, (y-p)^2)$. We also know that $Annotate(U)(t) = lift_{-P}(U)(t) = (1, -p, p^2)$. Since $R_{\bowtie}(\pi_{S_{R_{\bowtie}}}(t)) = (1, y, y^2)$, the annotation for $(Annotate(U) \bowtie R_{\bowtie})(t)$ is then $(1, -p, p^2) \times (1, y, y^2) = (1, y - p, p^2 + y^2 - 2py) = (1, y - p, (y-p)^2)$. □

Example 4. *The decision tree in Figure 3a has 3 leaf nodes whose predicates and predictions are:* $(\sigma_{D \le 1}, p = 2.5)$, $(\sigma_{D>1 \wedge C \le 1}, p = 1.5)$, *and* $(\sigma_{D>1 \wedge C>1}, p = 2)$. *The set of attributes involved with predicates are* $\mathbf{A} = \{C, D\}$. *The update relation is shown in Figure 3b. The semi-ring of residuals over join* $U \bowtie R_{\bowtie}$ *(with additional residual* $E = Y - P$ *column to help understanding) is shown in Figure 3d. It is easy to verify that the annotations in* R_ϵ *are the same as those in* $R \bowtie S \bowtie T \bowtie U$.

Challenges: The update relation can introduce cycles to the join graph; however *Message Passing* only applies to acyclic join graphs (e.g., Figure 3c shows cycle $A \to C \to D$). Standard hypertree decomposition [13, 43] removes cycles by joining the relations in a cycle and 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, this is not scalable for gradient boosting, because the cluster grows in size as update relations are added in each iteration. As the trees split using different attributes, we eventually converge to materializing the full join $R_{\bowtie}$.

We introduce two solutions that are sufficient for scalable factorized boosting in practical scenarios. The first exploits the N-to-1 relationships in snowflake schemas along the paths through the dimension tables: it rewrites decision tree leaf node predicates as

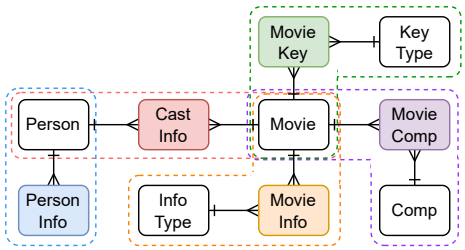


Figure 4: Clusters for IMDB dataset. Each cluster is enclosed by dotted lines and its fact table is filled.

a semi-join over the fact table rather than over $R_{\bowtie}$. The second extends the first solution to support galaxy schemas with multiple fact tables using a technique we call Clustered Predicate Trees.

4.3.1 Semi-join Selection for Fact Table: Snowflake schemas have a single tall fact table F and N-to-1 foreign key relationships along the path towards the dimension tables. In that case, we can rewrite predicates over dimension tables as semi-joins (predicates[5]) over the fact table. Consider a path $F - D_1 - ... - D_k$, and selection σ over D_i. Since D_{i-1} has a N-to-1 relationship with D_i,

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

where σ' is predicate $D_{i-1}.\mathcal{X} \in \pi_{\mathcal{X}}(\sigma(D_i))$ and $\mathcal{X}$ is the join keys between D_{i-1} and D_i. Therefore, predicates over the dimension tables can be recursively rewritten as semi-joins over the fact table. In these cases, F contains the attributes $\mathbf{A}$ in the update relation, and will not create any cycle [13].

To avoid accumulating more update relations in each boosting iteration, we replace F with $F \bowtie U$ in each iteration.

4.3.2 Clustered Predicate Tree: Rewriting selection as semi-join is not possible through a 1-to-N relationship. These are common in galaxy schema, which contains multiple fact tables. *Clustered Predicate Tree* (CPT) restricts each boosted tree to split on features that can be pushed to the same fact table. To do so, we cluster relations such that in each cluster, a single (fact) table F has N-to-1 relationships along the path to all other relations in the cluster. During training, the root decision tree node can be split on any feature, but the remaining splits are restricted to attributes in the same cluster. Although this can impact model accuracy, this lets us efficiently update residuals over join graphs that would otherwise be infeasible to train using existing techniques.

Example 5. *Figure 4 shows the IMDB [5] join graph. 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, then 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, API compatibility, and performance. We evaluate performance in the experiments, so focus on portability and compatibility below.

[5]We treat left semi-joins as filters over the left relation so its annotations don't change.

Portability: Our design deviates from prior factorized query execution works [22, 44, 46, 68], which build (fast) custom execution engines. In contrast, `JoinBoost` is implemented as a Python-based ML library that transparently generates SQL queries to the user's desired DBMS backend (Figure 5).

API Compatibility: `JoinBoost` exposes a similar Python API as `LightGBM` and `XGBoost`. In these libraries, the user first defines a training dataset, and passes it, along with training parameters (e.g., objective, metric, number of leaves), to a `train()` method. The `JoinBoost` user similarly defines the training dataset by initializing a join graph (the relations and join conditions between the relations), providing a database connection, and specifying the features and target variable. If the user only specifies the relations, `JoinBoost` will infer the join graph that covers those relations from the database schema, and raise an error if the graph is ambiguous (e.g., multiple foreign key references between relations) or it requires a cross-product. The user can freely define data transformation/pre-processing views and build the join graph atop those views. Finally, the user passes the training dataset and optional training parameters to `train()`. For consistency, `JoinBoost` also accepts the same training parameters as `LightGBM`.

Example 6. *Figure 5 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* $\mathbf{X}$ = `[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 a series of `CREATE TABLE` and `SELECT` SQL queries. Compared to in-DB systems like `MADLib` [35], `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 using `DuckDB`'s relational API [8].

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.

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

5.2 Architecture Overview

The *ML Compiler* translates the training algorithms for the desired model into semi-ring aggregation queries over the join graph, and returns a reference to the trained model. For instance, it issues queries to evaluate all possible splits for a given node, and chooses the best split. At this stage, the compiler treats the join graph as a large "wide" table; factorization is applied in a later step.

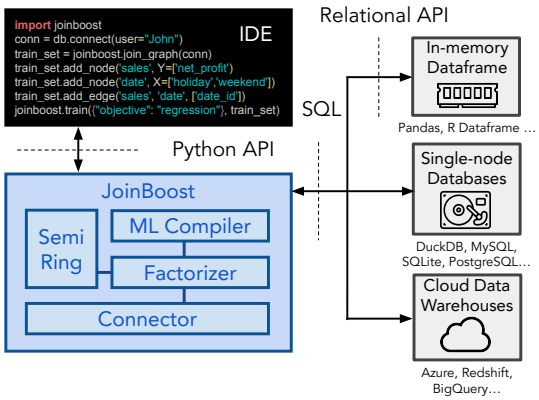


Figure 5: `JoinBoost` Architecture. `JoinBoost` translates its Python API calls into message passing algorithms that are executed as SQL queries on backend DBMSes and dataframes.

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 re-used in descendent nodes (Section 4.1) 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 4.3 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 the components of the semiring, 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 will directly `UPDATE` the fact table's semi-ring annotations. Let us start with a snowflake schema. Recall that each decision tree leaf node l logically corresponds to a separate join graph containing a set of messages (Figure 2), and let $l.\sigma$ be its predicate and $l.p$ be its prediction.

Also, the semi-join optimization in Section 4.3 translates predicates over $R_{\bowtie}$ (e.g., $l.\sigma$) into semi-joins between F and the 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 $l.\sigma$ and 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 j^{th} leaf in the decision tree, and $m_{i,j}$, $a_{i,j}$ be the i^{th} 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 [9] 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 a Azure VM, with Ubuntu 20.04.4, Intel(R) Xeon(R) Platinum 8370C CPU @ 2.80GHz, 16 cores, 128 GB memory, and an SSD.

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 i^{th} 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: $F' = F \bowtie_{\mathbf{A}} U$ as discussed in Section 4.3. `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` [65] is a

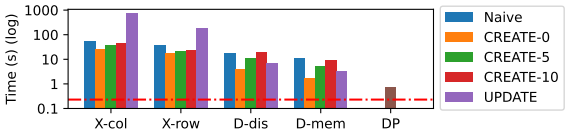


Figure 6: Residual update time (log) for different DBMSes. `X-col/row` are column/row-based backends of `DBMS-X`. `D-disk/mem` are disk/memory-based `DuckDB`. `DP` (Section 5.4) uses Pandas dataframe for the efficient update. The red horizontal line is the residual update time for `LightGBM`.

popular embedded column-oriented OLAP DBMS and is highly performant [10]. 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 6 shows that `Naive` incurs high materialization and join costs. `CREATE` is $\sim$ 2× faster for `DBMS-X` and $\sim$ 4× 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 $\sim 0.2s$. 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 [55] 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 failure. Compression is also unnecessary for the heavily updated columns. Therefore, we wish to evaluate residual updates when WAL, CC, and compression are turned off.

We simulate this in DuckDB by using their Relational API [8]. DuckDB supports direct access to Pandas dataframes [59], an in-memory data structure that stores columns in contiguous uncompressed C arrays. We store the fact table F as a dataframe, and use the relational API to join F with the rest of the tables in DuckDB. To update residuals, we found that creating a new column for the residuals yields the best performance and also avoids duplicating the other columns in F. Thus we first use DuckDB relational operators to compute (in multi-threads) the semi-ring annotations for updated residuals, whose result is stored in a NumPy array. We then replace the old column with it using the Pandas API (by changing the pointer). We call this combination `DP`, and it reduces residual updates to $0.72s$–competitive with `LightGBM` (Figure 6).

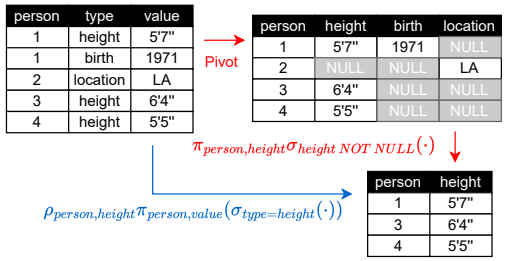


Figure 7: Pivot Transformation Optimizations over *Person_Info* in IMDB. Naively, the sparse pivot result is materialized, then each feature (e.g., height) is selected to evaluate the split criteria (red). The query could be rewritten as selection over type column by feature name, thus avoiding the materialization of sparse pivot table (blue).

A Final Note. While `DP` reduced the update bottleneck, joining DuckDB tables with Pandas dataframes does degrade aggregations, and thus training time, by $\sim$ 1.5× (Section 6.6). Ideally, columnar DBMSes can natively support this form of batch column-wise update at near-memory-bandwidth rates—for instance, based on copy-on-write semantics.

5.5 Additional Optimizations and Features

We apply a variety of optimizations to `JoinBoost` to improve performance and increase usability.

5.5.1 Inter-query Parallelism. Parallelism has been widely exploited by ML libraries. `LightGBM` implements parallelized sort, aggregation, and residual updates, and across split candidates. Similarly, `Sklearn` parallelizes across trees. For `JoinBoost`, most DBMSes provide intra-query parallelism. However, there are diminishing returns for any individual query or operation. Thus, `JoinBoost` aggressively parallelizes across queries as well. There are many parallelization opportunities across e.g., trees in a random forest, leaf nodes in a decision tree, candidates splits for a decision tree node, and messages to evaluate a candidate split. Of course, there are also dependencies between these queries: a message's query depends on its upstream messages, absorption depends on its incoming messages, tree nodes depend on queries from its ancestor nodes, and a gradient boosted tree depends on its preceding trees.

To this end, `JoinBoost` uses a simple scheduler. Each query Q tracks its dependent queries. When Q completes, it sets the ready bit for its dependent queries. If all of a query's dependencies are set, then it is added to a FIFO run queue. Empirically, we found that 4 threads worked best for intra-query parallelism, and the rest are allocated to inter-query parallelism. This inter-query optimization reduces the gradient boosting training time by 28% and random forest by 35% (Appendix C).

5.5.2 Data Transformations. Prior factorized learning work [46] 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 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 7 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. [25] 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.

5.5.3 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 [32] 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 results 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.

5.5.4 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.

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.

6 EXPERIMENTS

We now evaluate JoinBoost with factorized ML systems (LMFAO), in-database ML libraries (MADLib), and state-of-the-art ML libraries (XGBoost, LightGBM, and Sklearn). We evaluate their end-to-end training time with respect to the model size, the number of features, the types of joins, and the level and type of parallelization. We also evaluate the effectiveness of our optimizations.

Datasets. We use the Favorita [6] purchasing and sales forecast dataset (Figure 8), used in prior factorized learning works [68, 69]. Sales is the fact table and largest relation (2.7GB as CSV, 80M rows), and has N-to-1 relationships with the other dimension tables

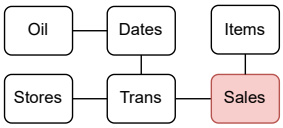


Figure 8: Favorita schema. Sales is the fact table.

(< $2MB$ each). There are 13 features. Additional experiments with TPC-DS at SF=10 can be found in Appendix C.

Preprocess. We dictionary encode strings into 32-bit unsigned integers following prior work [12, 66]. This avoids parsing errors in ML libraries like LightGBM.

Although Favorita (and TPC-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[6].

Workloads. 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 databases (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 databases, and the Python codes introduce negligible (<0.1s) overhead.

Methods. We evaluate JoinBoost with different database 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 can directly execute queries on the base database, whereas memory-based DuckDB (D-disk) first loads the database from disk. DP refers to the disk-based DuckDB backend using Pandas updates through the DuckDB's relational API. The ML libraries[7] (LightGBM, XGBoost, Sklearn) expect a single CSV as input, so incur the cost to materialize and export the join result (~$7GB$ 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: All experiments are conducted on an Azure VM, with Ubuntu 20.04.4, Intel(R) Xeon(R) Platinum 8370C CPU@2.80GHz, 16 cores, 128 GB memory, and an SSD.

6.1 Comparison With Factorized ML

LMFAO [68] is a fast factorized ML system that optimizes, compiles and executes batches of queries at a time. We verified with the authors that in their implementation, each batch of queries undergoes code generation, compilation, and data loading. Although this works well with a simple model like linear regression [69] that can be trained in a single batch and compilation can be done

[6]Favorita applies: $y = f_{item} log(f_{items}) + log(f_{oil}) - 10 f_{dates} - 10 f_{stores} + f^2_{trans}$
[7]For LightGBM and XGBoost, while their python APIs are widely used, it has memory issues (https://github.com/microsoft/LightGBM/issues/1032) so we use the CLI version.

offline, tree-based models generate a batch for each node split dynamically during run-time (based on parent node's best split). As a result, LMFAO has to incur repeated compilation and data loading overhead in end-to-end training times; such overheads could be costly [49, 54]. To report a fair comparison, we grow a tree stump (only split the root node) to execute a single query batch.

Figure 9a shows that *even for a tree stump*, LMFAO is >10× slower than JoinBoost due to compilation and loading costs. LMFAO is a custom C++ engine so its pure training time is ~1.2× faster. On the other hand, JoinBoost is portable to any DBMS.

To separate algorithmic vs implementation differences, we implement two variations of JoinBoost. Naive materializes the join result and does not use factorization. Batch implements LMFAO's core logical optimizations (Multi Root, Aggregate Push-down and Merge View) for decision tree—it performs *Message Passing* and re-uses messages among the batch of queries for each tree node, but doesn't share messages between tree nodes.

We train a decision tree (max depth=10) with best-first tree growth. The trained tree is balanced and has 1024 leaves. Figure 9b reports that Batch trains ~2× faster than Naive because of factorized learning and shared computations. JoinBoost is ~3× faster than Batch because JoinBoost caches and reuses half of the messages across tree nodes, and because 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 [13, 58] could help reduce message sizes, but the high overheads outweigh benefits [12].

6.2 Comparison With In-DB ML (MADLib)

MADLib [35] is a PostgreSQL extension that supports ML using user-defined types and functions. MADLib doesn't apply factorized learning, 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 9c shows that JoinBoost is ~16× faster than MADLib. This reproduces prior findings [68], and is likely due to the lack of factorized learning and an inefficient implementation.

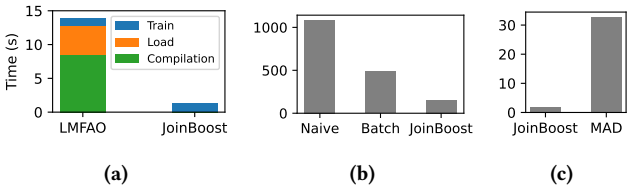
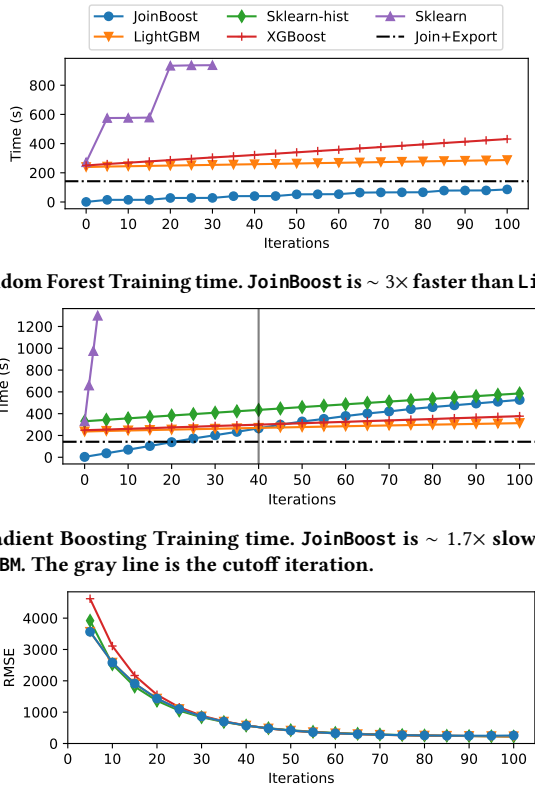


Figure 9: (a) Training time for decision stump. LMFAO is dominated by compilation and data loading. (b) JoinBoost's caching and work sharing improve over naive materialization and query batching variants. (c) Training time for decision tree. JoinBoost is ~ 16× faster than MADLib.



(a) Random Forest Training time. JoinBoost is ~ 3× faster than LightGBM.

(b) Gradient Boosting Training time. JoinBoost is ~ 1.7× slower than LightGBM. The gray line is the cutoff iteration.

(c) Gradient Boosting Accuracy. The final *RMSE* is nearly identical.

Figure 10: Gradient Boosting and Random Forest training time and model performance compared to the SOTA ML libraries. The dotted black line is the overhead of materializing and exporting the join for all ML libraries.

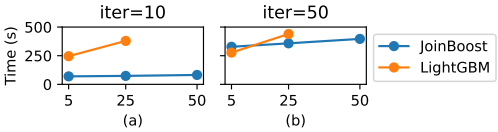


Figure 11: Gradient Boosting Training Time of 10^{th} (a) and 50^{th} (b) iteration when varying # of imputed features (x-axis). LightGBM runs out of memory when imputed 50 features.

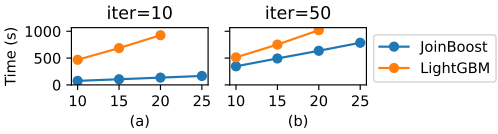


Figure 12: Gradient Boosting Training Time of 10^{th} (a) and 50^{th} (b) iteration. The X-axis varies TPC-DS SF (database size). LightGBM runs out of memory when $SF = 25$.

6.3 Comparison With ML Libraries

We now compare JoinBoost with SOTA ML libraries. LightGBM [44] and XGBoost [22] are state-of-the-art ML libraries for training tree-based models, and Sklearn [62] is a popular Python ML library.

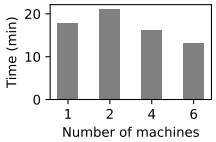


Figure 13: Decision Tree Training Time in Data Warehouse when varying the number of machines. Increasing the number of machines to 4 (6) reduces training time by 10% (25%).

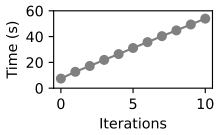


Figure 14: Gradient Boosting Training time over IMDB dataset with galaxy schemas. JoinBoost scales linearly and trains each boosted tree in ~5s.

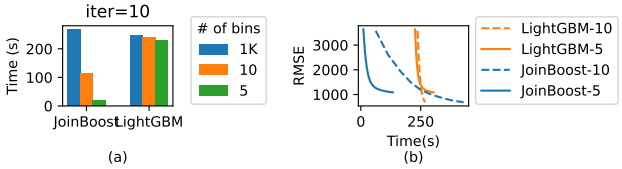


Figure 15: Histogram-based Gradient Boosting. (a) JoinBoost is much faster with fewer bins due to the smaller cuboid sizes. (b) Training time-accuracy trade-off. JoinBoost with $\#bin = 5$ pushes the Pareto frontier and reaches $RMSE = 1500$ in 20s.

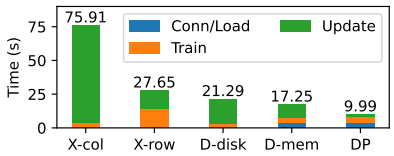


Figure 16: Breakdown of training one decision and updating the residuals for the Favorita dataset with different databases.

Sklearn implements standard and histogram-based gradient boosting with algorithms similar to LightGBM, so we report both implementations. We vary the number of bins to up to 1000 for LightGBM and XGBoost (each feature has at most 1000 unique values), and 255 for Sklearn (its limit). By default, we train gradient boosting and random forest with best-first growth and 8 max leaves per tree. The learning rate of gradient boosting is 0.1 and the random forest sampling rate without replacement is 10%. We vary the number of iterations from 1 to 100. JoinBoost uses D-disk for random forest and DP for gradient boosting as they are the most efficient (Section 6.6). The 0^{th} iteration reports a dummy model that uses the total average for prediction (along with the possible join materialization, exporting and loading costs).

Figure 10a shows random forest results. JoinBoost is ~3× 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 (~2500) for JoinBoost, LightGBM and XGBoost.

Figure 10b shows gradient boosting results. JoinBoost is ~1.7× slower than LightGBM and XGBoost, largely due to the residual update overhead and interop overhead between DuckDB tables and Pandas dataframes. However, JoinBoost is more efficient when training ≤40 trees (vertical line), and we see that the *RMSE* is within 10% of the final error. Recent work suggests that 40 iteration is sufficient when the learning rate is ≥0.1 [50]. Figure 10c shows that the model *RMSE* are nearly identical. LightGBM dominates the other ML libraries, so we primarily compare with it below.

6.4 Scalability

We now study scalability to # features, DB size, and join complexity.

 # Features. We train gradient boosting using Favorita and vary the number of features from 5 to 25, and report training time at the 10^{th} and 50^{th} iterations. Figure 11 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 ~80*GB* of intermediate results to disk (not plotted). JoinBoost scales linearly and benefits from DBMS support for efficient out-of-core query execution.

Single-node Scalability. We use TPC-DS (145 features) to vary the database size ($SF \in [10, 25]$). Figure 12 shows that both systems scale roughly linearly, but LightGBM runs out of memory at $SF = 25$.

Cloud-warehouse Scalability. We use a cloud data warehouse DW-X to train a decision tree with max depth 3 over TPC-DS with SF=1000, and study multi-machine scalability. Each machine has 74 cores, 300GB 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 4.3.2) to train gradient boosting on the IMDB dataset (Figure 4). Cast_Info is ~1*GB* and the total DB is 1.2*GB*. LightGBM cannot run because the join result is >1*TB*. JoinBoost scales linearly with the the number of iterations (Figure 14); it trains one tree and updates residuals in each cluster's fact table within ~5s.

6.5 Histogram-based Gradient Boosting

We vary the number of histogram bins to study when using the cuboid optimization (Section 5.5.3) is effective. For bin=5, 10, 1*K*, the cuboid contains ~3*M*, ~25*M*, and ~80*M* rows respectively.

Figure 15(a) shows that at iteration 10 and $bin = 1K$, the cuboid provides negative benefits. At $bin = 5$, 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 15(b)) JoinBoost-5 quickly finishes (~20s), but JoinBoost-10 takes longer to converge because the cuboid is larger and forgoes factorized learning. We leave exploring the interaction between cubes and factorized learning to future work.

6.6 Effect of DBMSes

We now train gradient boosting for 1 iteration to compare the different JoinBoost DBMS backends. Figure 16 breaks down the loading, training, and residual update costs. Columnar storage and execution are critical for *training*: X-col and DuckDB take $3.2-3.9s$ as opposed to $14.5s$ for X-row. DP reduces residual updates by >11× ($17.8s \rightarrow 1.5s$), but slows training by 50% ($3.2s \rightarrow 4.8s$) due to DuckDB-Pandas dataframe interop overhead.

Overall, D-disk is fastest for decision tree and random forest models because its query execution is comparable/faster than DBMS-X and avoids data loading overheads of in-memory DuckDB. DP is best for gradient boosting due to its fast residual updates.

7 CONCLUSION

JoinBoost is the first In-DB factorized learning system for tree models, including decision trees, random forests, and gradient boosting. 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] 2013. IMDB. https://www.imdb.com/interfaces/.
[6] 2017. Corporación Favorita Grocery Sales Forecasting. https://www.kaggle.com/c/favorita-grocery-sales-forecasting.
[7] 2021. 2021 Kaggle Data Science and Machine Learning Survey. https://www.kaggle.com/code/paultimothymooney/2021-kaggle-data-science-machine-learning-survey/notebook.
[8] 2021. Client APIs Overview. https://duckdb.org/docs/api/overview.
[9] 2021. (Technical Report) JoinBoost: In-Database Tree-Models In Action. https://anonymous.4open.science/r/JoinBoost-FBC4/technical/JoinBoost_tech.pdf.
[10] 2022. ClickBench: a Benchmark For Analytical Databases. https://benchmark.clickhouse.com/.
[11] 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.
[12] 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.
[13] 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.
[14] 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).
[15] 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).
[16] Alina Beygelzimer, Elad Hazan, Satyen Kale, and Haipeng Luo. 2015. Online gradient boosting. *Advances in neural information processing systems* 28 (2015).
[17] 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.
[18] Leo Breiman. 2001. Random forests. *Machine learning* 45, 1 (2001), 5–32.
[19] Leo Breiman, Jerome H Friedman, Richard A Olshen, and Charles J Stone. 2017. *Classification and regression trees*. Routledge.
[20] Bishop PRML Ch and Alireza Ghane. 1993. Sampling Methods. (1993).
[21] Lingjiao Chen, Arun Kumar, Jeffrey Naughton, and Jignesh M Patel. 2016. Towards linear algebra over normalized data. *arXiv preprint arXiv:1612.07448* (2016).
[22] 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.
[23] Rada Chirkova, Jun Yang, et al. 2011. Materialized views. *Foundations and Trends in Databases* 4, 4 (2011), 295–405.
[24] Francois Chollet et al. 2015. *Keras*. https://github.com/fchollet/keras
[25] 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.
[26] 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.
[27] Tony Duan, Avati Anand, Daisy Yi Ding, Khanh K Thai, Sanjay Basu, Andrew Ng, and Alejandro Schuler. 2020. Ngboost: Natural gradient boosting for probabilistic prediction. In *International Conference on Machine Learning*. PMLR, 2690–2700.
[28] 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.

[29] Walter D Fisher. 1958. On grouping for maximum homogeneity. *Journal of the American statistical Association* 53, 284 (1958), 789–798.

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

[31] Pierre Geurts, Damien Ernst, and Louis Wehenkel. 2006. Extremely randomized trees. *Machine learning* 63, 1 (2006), 3–42.

[32] 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.

[33] 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.

[34] 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).

[35] 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).

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

[37] Zezhou Huang and Eugene Wu. 2022. Calibration: A Simple Trick for Wide-table Delta Analytics. *arXiv e-prints* (2022).

[38] 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).

[39] Muhammad Idris, Martín Ugarte, and Stijn Vansummeren. 2017. The dynamic yannakakis algorithm: Compact and efficient query processing under updates. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1259–1274.

[40] 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).

[41] 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).

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

[43] 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.

[44] 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).

[45] 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.

[46] 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.

[47] Ron Kohavi and Chia-Hsin Li. 1995. Oblivious decision trees, graphs, and top-down pruning. In *IJCAI*. Citeseer, 1071–1079.

[48] 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.

[49] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-driven parallelism: A NUMA-aware query evaluation framework for the many-core age. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 743–754.

[50] Haihao Lu, Sai Praneeth Karimireddy, Natalia Ponomareva, and Vahab Mirrokni. 2020. Accelerating gradient boosting machines. In *International Conference on Artificial Intelligence and Statistics*. PMLR, 516–526.

[51] Llew Mason, Jonathan Baxter, Peter Bartlett, and Marcus Frean. 1999. Boosting algorithms as gradient descent. *Advances in neural information processing systems* 12 (1999).

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

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

[54] Thomas Neumann. 2011. Efficiently compiling efficient query plans for modern hardware. *Proceedings of the VLDB Endowment* 4, 9 (2011), 539–550.

[55] 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.

[56] 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.

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

[58] 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.

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

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

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

[62] 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.

[63] Wei Peng, Tim Coleman, and Lucas Mentch. 2019. Asymptotic distributions and rates of convergence for random forests via generalized u-statistics. *arXiv preprint arXiv:1905.10651* (2019).

[64] Liudmila Prokhorenkova, Gleb Gusev, Aleksandr Vorobev, Anna Veronika Dorogush, and Andrey Gulin. 2018. CatBoost: unbiased boosting with categorical features. *Advances in neural information processing systems* 31 (2018).

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

[66] 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.

[67] 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.

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

[75] 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.

[76] 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.

[77] 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`[8].

Aggregated Semi-ring: It has been shown that [13, 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;
```

[8] 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 [13]. 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:

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

B.2 Boosted Trees Preliminary

We provide the background of Boosted Tree based on [22, 44, 53].

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:** $((1, 0), ..., (1, 0))$	$((h_1^1, g_1^1), ..., (h_1^k, g_1^k)) + ((h_2^1, g_2^1), ..., (h_2^k, g_2^k))$ $= ((h_1^1 + h_2^1, g_1^1 + g_2^1), ..., (h_1^k + h_2^k, g_1^k + g_2^k))$ $((h_1^1, g_1^1), ..., (h_1^k, g_1^k)) \times ((h_2^1, g_2^1), ..., (h_2^k, g_2^k))$ $= ((h_1^1 h_2^1, g_1^1 h_2^1 + g_2^1 h_1^1), ..., (h_1^k h_2^k, g_1^k h_2^k + g_2^k h_1^k))$	$((h^1(t), g^1(t)), ..., (h^k(t), g^k(t)))$

Table 2: Gradient Semi-rings for gradient boosting.

Task	Loss function	Gradient $g(\cdot)$	Hessian $h(\cdot)$	Prediction P
Regression	L2/rmse: $(\epsilon)^2$	ϵ	1	$mean(\mathcal{E})$
	L1/mae: $\lvert\epsilon\rvert$	$sign(\epsilon)$	1	$median(\mathcal{E})$
	Huber Loss: $\begin{cases} 0.5\epsilon^2, \lvert\epsilon\rvert \le \delta \\ \delta(\lvert\epsilon\rvert - 0.5\delta), else \end{cases}$	$\begin{cases} \epsilon, \lvert\epsilon\rvert \le \delta \\ \delta \cdot sign(\epsilon), else \end{cases}$	1	p^*
	Fair Loss: $c\lvert\epsilon\rvert - c^2 log(\lvert\epsilon\rvert/c + 1)$	$c\epsilon/(\lvert\epsilon\rvert + c)$	$c^2/(\lvert\epsilon\rvert + c)^2$	p^*
	Poisson Loss: $e^p - yp$	$e^p - y$	e^p	p^*
	Quantile Loss: $\begin{cases} (\alpha - 1)\epsilon, \epsilon < 0 \\ \alpha\epsilon, \epsilon \ge 0 \end{cases}$	$\begin{cases} \alpha - 1, \epsilon < 0 \\ \alpha, \epsilon \ge 0 \end{cases}$	1	$pctl_\alpha(\mathcal{E})$
	mape: $\lvert y - p\rvert / max(1, \lvert y\rvert)$	$\frac{sign(y-p)}{max(1, \lvert y\rvert)}$	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:

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

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:

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

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) [9].

C OTHER EXPERIMENTS

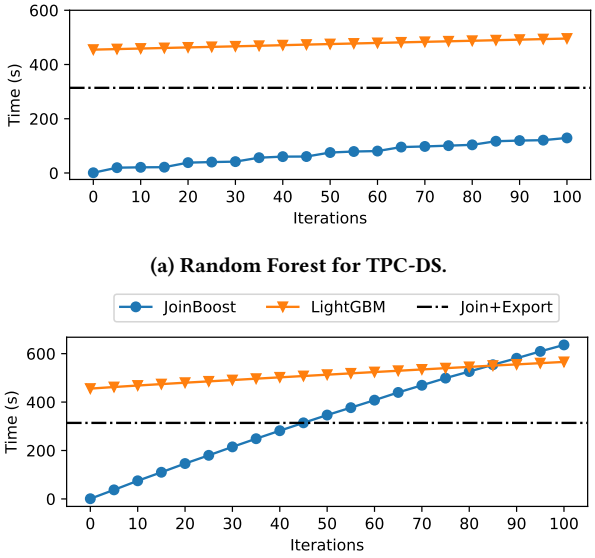
C.1 Comparison with `LightGBM`

Figure 17 shows the experiment results for TPC-DS with SF=10. `JoinBoost` is ~ 3× faster than `LightGBM` for Random Forest, and ~ 3× slower than `LightGBM` for Random Forest.

C.2 Parallelization

We use `JoinBoost` with (`para`) and without (`w/o`) inter-query parallelism, but always with intra-query optimization (4 threads per query for `para` and 16 threads per query for `w/o`), to train 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 Random Forest, the tree-wise parallelism reduces the training time more significantly by 35%.

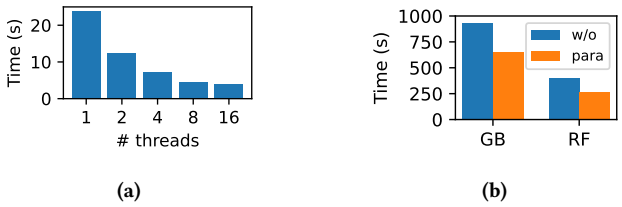
[9] https://github.com/microsoft/LightGBM/pull/175



(a) Random Forest for TPC-DS.

(b) Gradient Boosting for TPC-DS.

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.



(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.

D OTHER OPTIMIZATIONS

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 $\sigma_l(L)$.