LEAP: A Low-cost Spark SQL Query Optimizer using Pairwise Comparison

Junhao Ye
Zhejiang University
junhao_ye@zju.edu.cn

Jiahui Li
Zhejiang University
li.jiahui@zju.edu.cn

Lu Chen
Zhejiang University
luchen@zju.edu.cn

Yuren Mao
Zhejiang University & Zhejiang Key Laboratory of Big Data Intelligent Computing
yuren.mao@zju.edu.cn

Yunjun Gao
Zhejiang University & Zhejiang Key Laboratory of Big Data Intelligent Computing
gaoyj@zju.edu.cn

Tianyi Li
Aalborg University
tianyi@cs.aau.dk

ABSTRACT

Selecting a good execution plan can significantly improve the query efficiency of Spark SQL. Several machine learning-based techniques have been proposed to select good execution plans for DBMS, but none of them perform well on Spark SQL due to the following issues. (1) Limited compatibility with Spark SQL: these approaches rely on physical operator enumeration, while Spark SQL doesn't support it; (2) Unreliable cost estimation: they often select execution plans with poor performance due to inaccurate cost estimation; (3) Time-consuming plan enumeration: they take much time to generate a large number of candidate execution plans in Spark SQL. To overcome these issues, in this paper, we propose LEAP, the first learned query optimizer tailored for Spark SQL, which can be integrated seamlessly into Spark SQL and solves the compatibility issue. Also, to avoid the unreliable cost value estimation, LEAP selects execution plans with an estimation-free method, which directly performs comparisons between the plans. Furthermore, LEAP employs an efficient progressive plan enumeration algorithm with pruning techniques to find better plans with fewer enumerations. Extensive experiments on three public benchmarks show the effectiveness of LEAP. It reduces the end-to-end execution time of the native optimizer by up to 54% and other learned methods by up to 94%.

PVLDB Reference Format:
Junhao Ye, Jiahui Li, Lu Chen, Yuren Mao, Yunjun Gao, and Tianyi Li. LEAP: A Low-cost Spark SQL Query Optimizer using Pairwise Comparison. PVLDB, 18(3): 675 - 687, 2024.
doi:10.14778/3712221.3712234

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/HuashiSCNU0303/LEAP.

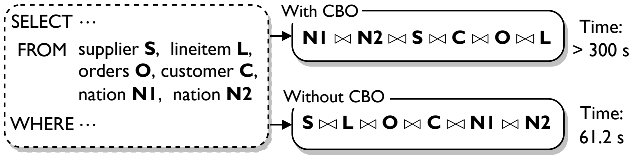


Figure 1: The join plans and their corresponding execution time of TPC-H Q7, with or without CBO.

1 INTRODUCTION

With the rise of big data, Spark SQL [5] has become increasingly popular for processing and analyzing large-scale data in a distributed environment. For an input query Q, Spark SQL generates an execution plan (or join plan) to specify the sequence in which tables are joined and how data operations are executed. Given the substantial data volumes and extended run times of OLAP workloads, selecting efficient execution plans for queries on Spark SQL is crucial for query optimization. Different execution plans for the same query can significantly vary execution times. Historically, Spark SQL's optimization has relied on its Cost-based Optimization (CBO) module, which utilizes column statistics like the number of distinct values (NDV) and histograms to estimate the cost of various execution plans and select the most cost-effective one. However, these estimates are often inaccurate, leading to the selection of poor plans by Spark SQL's CBO. For example, for TPC-H Q7, enabling CBO leads to a query execution time exceeding 300s, which is much more than 61.2s without CBO, as shown in Figure 1.

To further improve query performance, machine-learning-based techniques have become viable approaches in the DBMS field, which mainly focuses on two strategies: learned cardinality estimation and learned optimizer. Learned cardinality estimation methods use machine learning models to predict the output row count (i.e., cardinality) of execution plans, thereby refining the native optimizer's cost estimates to generate better plans. Learned optimizers often use machine learning models to directly estimate costs for entire execution plans, and select the most cost-effective one from a set of promising candidate plans. Notable techniques include plan-constructor methods [9, 23, 43] and plan-steerer methods [10, 22, 45, 49].

However, applying these DBMS-based methods to Spark SQL is challenging due to the following reasons.

Limited compatibility with Spark SQL. Many existing methods can't fully optimize Spark SQL queries due to their limited compatibility with Spark SQL system. For example, hint-set-based

Proceedings of the VLDB Endowment, Vol. 18, No. 3 ISSN 2150-8097.
doi:10.14778/3712221.3712234

methods [22, 42] can optimize physical operator selection but not join orders in Spark SQL, as Spark SQL determines join order based on logical costs related to output cardinalities, unlike traditional DBMSs, which consider physical operators in their plan cost. Thus, disabling physical operators generates candidate plans of the same join order. Also, Lero [49] generates join plans by scaling the native cardinality estimates. That may lead to join plans with large table broadcasts or shuffles, which greatly increases network or I/O costs, causing performance drops or even execution failures. Moreover, existing methods often require modifications to Spark SQL's source code, which needs extensive architectural knowledge and extra debugging and maintenance costs.

Unreliable cost estimation. Existing methods often select join plans with poor performance due to inaccurate cardinality or cost estimations from learned estimators. These inaccuracies stem from predicting continuous outputs, making them sensitive to outliers and noise, especially with limited training data. For example, learned cardinality estimators can produce estimates deviating from actual values by tens of times, resulting in join plans with execution times up to 8 to 10 times longer than the optimal ones [33].

Time-consuming plan enumeration. The enumeration space for a query with n tables can reach at least $O(n!)$, making it time-consuming for machine learning models to estimate plan cost or output cardinalities. Additionally, generating an execution plan using Spark SQL's native optimizer is slower compared to DBMS. Creating multiple candidate plans, as required by plan-steerer methods, also demands substantial time.

To overcome these issues, we propose a new learned optimizer tailored for Spark SQL, named LEAP. It introduces the following three modules to tackle the above three challenges respectively:

Optimization framework. To enhance compatibility, we develop an optimization framework to align with Spark SQL system. Spark SQL optimizes queries in two stages: first, it enumerates logical join orders to find the one with the minimum logical cost (related to output cardinality); then, it assigns physical operators based on heuristic rules without enumerating them. Thus, LEAP introduces two modules, Join Plan Enumerator and Join Operator Selector, to optimize join order and physical operator selection respectively. For a query Q, Join Plan Enumerator processes it to generate a cost-effective join plan P. Join Operator Selector then assigns physical operators to each join operation in join plan P. These modules can be integrated into existing workflows seamlessly without modifications to Spark SQL's source code, thus improving the usability.

Learned comparator. To avoid unreliable cost estimation, we design a learned comparator $C(P_1, P_2)$ to evaluate two join plans P_1 and P_2. If P_1 is better, $C(P_1, P_2) = 0$, and vice versa. This pairwise comparison approach allows us to identify the best plan among candidates more effectively than regression-based methods, as the optimizer only needs to predict relative costs of join plans to choose one, not their exact costs. Compared to existing comparators [10, 42, 49], our comparator improves accuracy by using predicate information and data features. Also, our comparator uses sequence model to process plans, which may generate better plans than tree models with limited training data [48]. Moreover, our model does not require multiple join plans for the same query as training data; it can be trained directly using accumulated historical query data, significantly reducing computational resource demands.

Enumeration algorithm. To reduce enumeration space, we propose an efficient progressive plan enumeration algorithm to generate a low-cost join plan for a given query Q. The algorithm progressively adds a new table to existing sub-plans to generate a left-deep tree plan. While left-deep plans are simpler, bushy tree plans are preferred in distributed systems like Spark SQL due to their ability to parallelize join operations, though their search space is larger. Thus, we integrate bushy-tree plan enumeration within the search for left-deep tree plans. Specifically, during each iteration, when encountering a sub-plan P_1, the algorithm attempts to generate a bushy tree plan that includes P_1, that is, it recursively generates a join plan P_2 for the remaining tables, and then joins P_1 with P_2. Additionally, the algorithm uses the cost of full join plans to prune the unpromising sub-plans in subsequent searches.

Our contributions are summarized as follows.

- We propose LEAP, the first learned query optimizer tailored for Spark SQL, which is **simple yet effective**. It consists of two modules: Join Plan Enumerator and Join Operator Selector, to optimize join order and physical operator selection, respectively, which can be seamlessly integrated into existing workflows.
- We develop a Learned Comparator to evaluate the relative cost of two join plans rather than their exact cost, which helps to select better join plans. It provides more accurate comparisons with predicate information and data features.
- We propose an efficient progressive plan enumeration algorithm to reduce the enumeration space, which incorporates beam search strategy and pruning techniques.
- We conduct extensive experiments on three public benchmarks. LEAP reduces the native optimizer's end-to-end execution time by up to 54% and other learned methods by up to 94%.

We organize this paper as follows. Section 2 introduces Spark SQL's optimization process. Section 3 presents an overview of LEAP. Sections 4, 5, 6 describe three main components of LEAP respectively. Section 7 analyzes the experimental results. Section 8 displays the related works. Finally, Section 9 concludes the paper.

2 PRELIMINARY

In this section, we provide an introduction to the join plan and the optimization process of Spark SQL.

2.1 Join Plan

Given a query Q composed of n tables $\mathcal{T} = \{T_1, T_2, T_3, \ldots, T_n\}$, a join plan P is a tree consisting of these tables to specify the order of operations. In P, leaf nodes are filter operations on tables, while non-leaf nodes represent logical join operations (e.g., Inner Join ($\bowtie$)). We denote the number of output rows of P as its output cardinality $Card(P)$, and refer to the tables involved in P as $\mathcal{T}(P)$. Additionally, any subtree within P is defined as a sub-plan.

There are left-deep tree plans and bushy tree plans. In a left-deep tree plan, the left child of any join operation is a subtree, and the right child must be a leaf node, as shown in P_1 in Figure 2. It allows only sequential execution of join operations, with a $O(n!)$ search space. In contrast, bushy tree plans allow both children of any join operation to be subtrees, as shown in P_2 in Figure 2. This structure enables parallel execution of join operations. The complexity of possible join plan trees grows significantly to $\frac{(2n-2)!}{(n-1)!}$ [26].

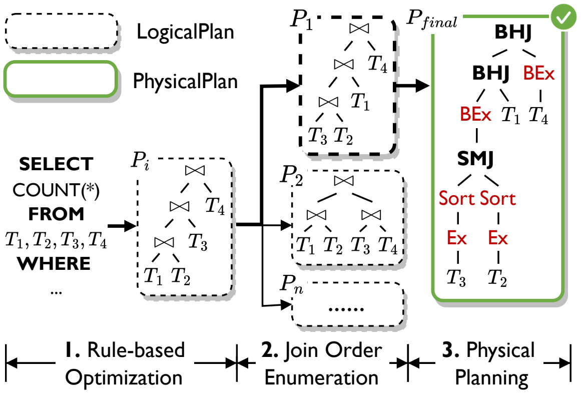


Figure 2: An overview of Spark SQL's optimization process. BHJ and SMJ denote physical join operators, while the red ones are additional physical operators.

2.2 Optimization Process of Spark SQL

In Spark SQL, there are two types of execution plans: LogicalPlans and PhysicalPlans, as shown in Figure 2. LogicalPlan corresponds to the "join plan" in Section 2.1, which only contains logical operations. PhysicalPlan replaces these logical operations with concrete physical operators and adds additional ones (e.g., Exchange). For a query Q, Spark SQL uses the following three stages to first transform Q to an optimized LogicalPlan P_1 and then generate a PhysicalPlan P_{final} for execution, as shown in Figure 2. Note that LEAP only operates on LogicalPlan.

2.2.1 Rule-based Optimization. In this stage, native optimizer transforms query Q into an initial LogicalPlan P_i. It converts Q into a LogicalPlan P_i with a parser and an analyzer, and then optimizes P_i using a series of optimization rules (e.g., predicate pushdown).

2.2.2 Join Order Enumeration. In this stage, native optimizer selects a low-cost LogicalPlan P_1 by reordering the Inner Join operations in P_i. The cost of a LogicalPlan is measured by the estimated cardinality and output size, without considering physical operators or resource consumption [7]. Based on the tables $\mathcal{T}$ in query Q, it uses a dynamic programming algorithm to enumerate different join plans from bottom to up. Specifically, the enumeration begins with 1-table sub-plans and incrementally constructs i-table sub-plans from existing j-table sub-plans (where $1 \leq j \leq i-1$). It terminates after sub-plans containing all tables are generated. During the enumeration, native optimizer utilizes the CBO module to estimate the cost of each LogicalPlan, and selects the one P_1 with the lowest cost among candidates P_1, P_2, ..., P_n. The order of join operations significantly impacts query performance. A suitable LogicalPlan can reduce the size of intermediate tables, thereby reducing the need of time and resource of intermediate join operations.

2.2.3 Physical Planning. In this stage, native optimizer assigns physical operators to each join operation in P_1 based on heuristic rules, and then inserts additional physical operators (e.g., Exchange, Sort) accordingly to generate the PhysicalPlan P_{final} for execution. When assigning physical join operators, native optimizer uses the CBO module to estimate the output size of each sub-plan in P_1. If the output size of a sub-plan P_s is less than the parameter *spark.sql.autoBroadcastJoinThreshold* (or BJT for short), the output of sub-plan P_s is considered suitable for broadcasting. For each join operation, if the output of sub-plan on either side is suitable for broadcasting, a high-performance Broadcast Hash Join (BHJ) is used. Otherwise, a Sort Merge Join (SMJ) or Shuffled Hash Join (SHJ) is employed. Using BHJ properly can avoid shuffling large tables and reduce query time. However, using BHJ inappropriately, such as broadcasting large tables due to output size estimation errors, can lead to increased execution times and even query failures.

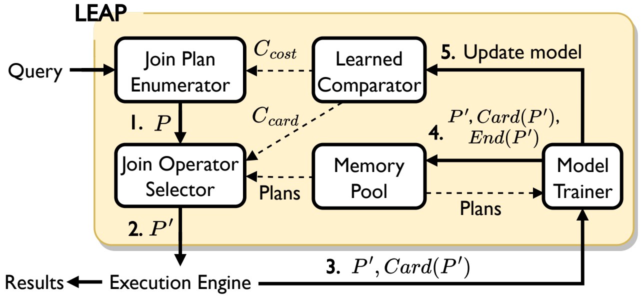


Figure 3: An overview of LEAP.

3 SYSTEM OVERVIEW

In this section, we present an overview of LEAP, and discuss its ability to adapt to different scenarios. It's used for join optimization, as LEAP is designed to replace Spark SQL CBO, which mainly focuses on join order enumeration and join operator selection.

3.1 System Framework

Figure 3 shows LEAP's framework with five components: **Memory Pool**, **Learned Comparator**, **Join Plan Enumerator**, **Join Operator Selector**, and **Model Trainer**. For a query Q, Join Plan Enumerator generates a join plan P using Learned Comparator C_{cost}. Then, Join Operator Selector assigns physical operators to the join operations in P using Learned Comparator C_{card}, resulting in a final join plan P'. Subsequently, Spark SQL executes the join plan P'. After execution, Model Trainer collects join plan P' and its output cardinalities $Card(P')$ and saves them to Memory Pool. It also periodically updates Learned Comparators C_{cost} and C_{card}.

Memory Pool. It saves historical join plans in memory for the update of Learned Comparators. It's a key-value storage where the key is $Card(P)$, and the value is a triplet $(P, Card(P), End(P))$, where $P, Card(P), End(P)$ denote the join plan, its output cardinality, and its finish timestamp respectively. Memory Pool is updated after the execution of each query.

Learned Comparator $C(P_1, P_2)$. It compares two join plans P_1 and P_2 in terms of a performance-related metric $L(P)$ (e.g., latency, cardinality). Formally, $C(P_1, P_2) = 1$ when $L(P_1) \geq L(P_2)$, and 0 otherwise. When $C(P_1, P_2) = 1$, P_1 is worse than P_2. We train two Learned Comparators C_{cost} and C_{card} using different $L(P)$ (i.e., the execution cost and output cardinality).

Join Plan Enumerator. It generates a low-cost join plan for query Q. It employs an enumeration algorithm to iteratively search for left-deep tree plans, and at each iteration step, it attempts to generate bushy tree plans. Finally, it selects the join plan with the lowest cost from all candidate plans by pairwise comparisons using C_{cost}.

Join Operator Selector. It assigns appropriate physical operators for each join operation in the join plan P. For simplicity, we only

focus on whether to use Broadcast Join (i.e., Broadcast Hash Join or Broadcast Nested Loop Join), which is sufficient for acceptable performance improvement. First, it selects some plans with known cardinalities from Memory Pool. It then compares these plans with each sub-plan P_s in join plan P using C_{card}, to determine whether the output of each sub-plan P_s can be broadcast, and selects the appropriate physical operator for each join operation accordingly.
Model Trainer. It updates the Learned Comparators based on the data in Memory Pool. After query executions, it collects the join plan P' and its output cardinality $Card(P')$, and saves the tuple $(P', Card(P'), End(P'))$ into Memory Pool. Also, it regularly clears old data from Memory Pool and re-trains the Learned Comparators C_{cost} and C_{card} using the accumulated new data.

3.2 Discussion

Here we discuss LEAP's ability to adapt to various scenarios.
Cold-start scenarios. When there are no training plans for a new workload, we can initialize LEAP's Learned Comparator with Spark SQL's native estimates similar to [43, 49]. Specifically, we enumerate sub-plans using Spark SQL's native optimizer for each training query, and train LEAP's comparator using such plans, with native cardinality estimates as labels. Also, as reported in Section 7.2.4, LEAP performs better than native optimizer even with few training queries, which can be quickly collected in production.
Compatibility with other systems. While LEAP is initially designed for Spark SQL, it can support other big data query systems. In other systems, LEAP rewrites SQL queries based on the optimized join order and specifies physical operators using hints, if supported. LEAP first converts the input SQL query into a system-agnostic logical plan P, containing only logical operations (e.g., joins and filters) and regardless of physical operators. It then optimizes join order and physical operators in P to produce an optimized plan P'. Finally, LEAP rewrites the query and organizes the physical operator hints based on P', and sends the rewritten query to the target system for execution. As reported in Section 7.2.3, LEAP also achieves performance improvement in Presto and Apache Doris.
Compatibility with other queries. LEAP can support query types beyond Select-Project-Join (SPJ) queries. It can be extended to handle other operators (e.g., Aggregate, Intersect, Outer Join) by modifying the one-hot encoding of logical operators in Section 4.1.2. At present, LEAP does not support queries with predicate sub-queries (e.g., sub-queries in IN or EXISTS), since Spark SQL decorrelates them after join order enumeration, preventing us from collecting training labels. We plan to address this limitation in future work.

4 LEARNED COMPARATOR

Learned Comparator $C(P_1, P_2)$ compares the performance-related metric $L(P)$ of join plans P_1 and P_2. If $L(P_1) \geq L(P_2)$, $C(P_1, P_2) = 1$, and 0 otherwise.

4.1 Model Design

Learned Comparator processes join plan trees P_1 and P_2 by first flattening them into node sequences and transforming each node into a vector representation. These vectors are then aggregated using LSTM to form representations for the entire trees of P_1 and P_2, which are subsequently compared to produce the final result $C(P_1, P_2)$, as shown in Figure 4.

4.1.1 Plan tree linearization. It transforms the join plan tree P into a node sequence $Seq(P)$. Using a direct node traversal sequence would lose structural information. To preserve this, we linearize the tree into a SBT (Structure-based Traversal) node sequence, inspired by code representation techniques [16]. We use a preorder traversal but add a terminal node after each subtree to mark its end. For instance, if a subtree's root node is 'Inner Join', a terminal node labeled ') Inner Join' follows its traversal.

4.1.2 Plan node representation. It converts each join plan tree node $N \in Seq(P)$ into a low-dimensional vector representation $Emb(N)$. As shown in Figure 4, $Emb(N)$ is the concatenation of following four parts: (1) a one-hot encoding of the logical operator on N (e.g., Filter, Inner Join, Left Join), (2) 0/1 encoding of tables touched by N, (3) the row count of chosen tables, and (4) the predicate embedding of the predicates in N. Row counts undergo logarithmic transformation and min-max scaling to scale values within [0, 1], and the details about predicate embeddings are in Section 4.1.3.

4.1.3 Predicate representation. It converts the predicates in join plan tree node N into vector representations. Predicates can also be viewed as a tree, where leaf nodes are atomic predicates f (such as $table.id = 7$), and non-leaf nodes are boolean operators (AND, OR, NOT). We first linearize the predicate tree into an SBT node sequence as in Section 4.1.1. Then, we extract features for each node (atomic predicate f or boolean operator), and finally aggregate the features of all nodes through a LSTM to get the final predicate embedding.
Atomic predicate featurization. An atomic predicate f (such as $table.id = 7$) consists of a column $Col(f)$, an operator $Op(f)$ (e.g., >, <, IN, LIKE), and an operand $Val(f)$. Its feature $Emb(f)$ is the concatenation of the following five parts, as shown in Equation 1,

$$Emb(f) = [Col(f)_{id}\ Op(f)_{id}\ Val(f)\ Hist(f)\ Sel(f)] \quad (1)$$

where $Col(f)_{id}$ and $Op(f)_{id}$ are the one-hot encoding of columns and operators, and $Val(f)$ is the normalized value of the operand. Histogram embedding $Hist(f)$ and selectivity embedding $Sel(f)$ serve as data features, and we discuss their details below.

(1) **Histogram** $Hist(f)$**.** It featurizes the histogram of column $Col(f)$, which represents the data distribution by dividing data into intervals (a.k.a bins). Each bin B_i's width $W(B_i)$ (interval length) may vary while the height $H(B_i)$ (number of elements in B_i) is uniform in Spark SQL's equi-height histograms. For numeric column $Col(f)$, we extract its equi-height histogram from Spark SQL, and $Hist(f)$ is the concatenation of boundary values of each bin. Histogram features help Comparator understand data distributions to handle skewed distributions.
(2) **Selectivity** $Sel(f)$**.** It represents the estimated selectivity of f. We estimate the selectivity by diving the column $Col(f)$ into m bins to estimate the number of qualified elements n_i in the i-th bin, and then summing up n_i as f's overall selectivity ($1 \leq i \leq m$). Thus, $Sel(f)$ is the concatenation of n_i and f's overall selectivity. We discuss how to compute n_i as below.
    - If $Col(f)$ is numeric, we use its histogram as m bins to estimate n_i. For each bin B_i in the histogram, we determine

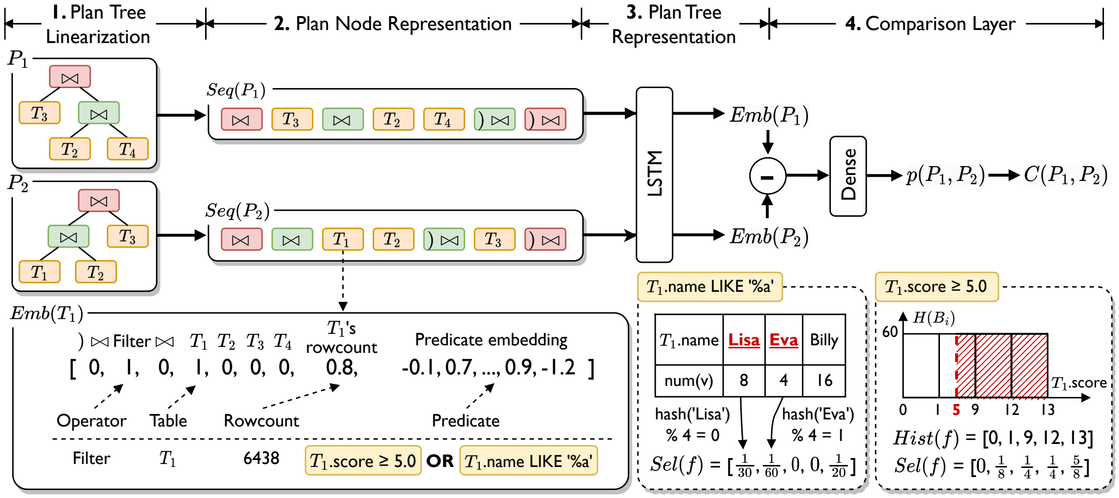


Figure 4: Our comparator model. ⋈ represents Inner Join, while) ⋈ represents the end of the subtree rooted at a ⋈ node.

its intersection with f's range $f \cap B_i$, and use the intersect width $|f \cap B_i|$ to calculate $n_i = \frac{|f \cap B_i|}{W(B_i)} \times H(B_i)$.

- If $Col(f)$ is string-based, histograms are not available. Thus, we divide $Col(f)$ into m bins using the hash values of the elements. We build a Most Common Value (MCV) list that identifies the most frequent values v and their counts $num(v)$ in $Col(f)$, using this list to estimate n_i. For predicate f, we identify values $V = \{v_1, v_2, \ldots, v_n\}$ from the MCV list that match the predicate f. For each matching value $v \in V$, we use its hash value $hash(v)$ to allocate it to the i-th bin, and update the corresponding n_i with $num(v)$, and thus, $n_i = \sum_{v \in V: hash(v)\%m=i} num(v)$. This approach can estimate selectivity for complex string predicates such as IN and LIKE, which are not widely supported in previous works. This approach works well on string columns with any distribution. For an infrequent or unique value v, we estimate its count $num(v)$ using NDV. Our MCV list contains up to 1000 frequent values per column, and NDV estimates are accurate for values outside this list.

We concatenate the n_i and the sum of n_i as the selectivity embedding $Sel(f)$. To normalize the values, we divide each value by the number of total elements $|Col(f)|$ to scale within $[0, 1]$. Specifically, $Sel(f) = \left[n_1, n_2, ..., n_m, \sum_{1 \le i \le m} n_i\right] / |Col(f)|$. For example, for a numeric predicate $f_1 : T_1.score \ge 5.0$ in Figure 4, the histogram of $T_1.score$ has 4 bins, each with a height $H(B_i) = 60$. f_1 selects no element in the first bin, half in the second, and all the elements in the third and fourth bins. Thus, $n_i = [0, 30, 60, 60]$, leading to $Sel(f_1) = \frac{1}{240}[0, 30, 60, 60, 150]$. Also, for a string predicate $f_2 : T_1.name$ LIKE '%a' in Figure 4, we set $m = 4$, and the MCV List of $T_1.name$ records three common values 'Lisa', 'Eva' and 'Billy' in this column and their occurrence counts. Since only 'Lisa', 'Eva' end with 'a' in MCV List, we compute n_i based on the two values, and get $n_i = [8, 4, 0, 0]$. Thus, $Sel(f_2) = \frac{1}{240}[8, 4, 0, 0, 12]$.

Boolean operator featurization. We use one-hot encoding for each type of boolean operator as their features.

4.1.4 Plan tree representation. It generates the vector representation $Emb(P)$ of the entire join plan P. It employs a LSTM network to aggregate the representation $Emb(N)$ of each node $N \in Seq(P)$, and the final hidden state of LSTM network is regarded as $Emb(P)$.

4.1.5 Comparison layer. It compares the vector representations of join plans P_1 and P_2 (i.e., $Emb(P_1)$ and $Emb(P_2)$), and outputs a binary label (0 or 1) based on their comparison. We use $Emb(P_1) - Emb(P_2)$ as the final representation of the pair (P_1, P_2), and it's then passed through a linear layer to generate an output logit. The logit is then transformed by a sigmoid function $\sigma(x) = 1/(1 + e^{-x})$ to yield a probability value $p(P_1, P_2) \in (0, 1)$, as shown in Equation 2.

$$p(P_1, P_2) = \sigma(\boldsymbol{W}(Emb(P_1) - Emb(P_2)) + \boldsymbol{b}) \qquad (2)$$

where $\boldsymbol{W}$ and $\boldsymbol{b}$ are learnable parameters.

Finally, the binary indicator $C(P_1, P_2)$ is defined as a step function via the probability $p(P_1, P_2)$. Specifically, if $p(P_1, P_2) \ge 0.5$, then $C(P_1, P_2) = 1$, which indicates $L(P_1) \ge L(P_2)$, and vice versa.

4.2 Model Training

We randomly generate some training queries $\boldsymbol{Q}$, and execute these queries on Spark SQL. Suppose the training query $Q \in \boldsymbol{Q}$ is executed using join plan P, we collect the corresponding $L(P)$ to form a set of join plans $\mathcal{P}$. Then, we generate plan pairs from $\mathcal{P}$ to build the training data $\mathcal{D}$. For each plan pair $(P_1, P_2) \in \mathcal{P}$, if $L(P_1) \ge L(P_2)$, we assign a label $l(P_1, P_2) = 1$; otherwise, $l(P_1, P_2) = 0$. Then, we include the triplet $(P_1, P_2, l(P_1, P_2))$ in the training dataset $\mathcal{D}$. We train comparator models using $\mathcal{D}$ with binary cross entropy loss.

4.3 Different Comparators

We train two Learned Comparators C_{cost} and C_{card} with different $L(P)$, to compare the execution cost and output cardinality of two join plans. These two comparators are used in Join Plan Enumerator and Join Operator Selector, as detailed in Sections 5 and 6.

Cost comparator C_{cost}. It compares the execution costs of join plans P_1 and P_2 by using $Cost(P)$ as $L(P)$. Assume that join plan P is a Γ-layer tree, we define its execution cost $Cost(P)$ as the sum of the maximum output cardinality of join nodes at each layer. Formally,

Algorithm 1: findBestPlan($\mathcal{T}, k$)

```
   Input  :Tables $\mathcal{T}$, beam width $k$
   Output :A cheapest join plan composed of $\mathcal{T}$'s tables
1  if $|\mathcal{T}| \leq 2$ then return a join plan composed of $\mathcal{T}$'s tables;
2  $\mathcal{P}_t \leftarrow$ findTopK({createJoin($T_i, T_j$), $\forall T_i, T_j \in \mathcal{T} (i < j)$}, $k$);
3  $P_o \leftarrow$ None ;              /* to store the optimal plan */
4  for $i \leftarrow 2$ to $|\mathcal{T}| - 1$ do
       // Quasi-bushy tree search
5      for $P_1 \in \mathcal{P}_t$ do
6          if canBeBroadcast($P_1$) then
7              $P_2 \leftarrow$ findBestPlan($\mathcal{T} \setminus \mathcal{T}(P_1), k$);
8              $P_b \leftarrow$ createJoin($P_1, P_2$);
9              if $C_{cost}(P_o, P_b) = 1$ then $P_o \leftarrow P_b$;
       // Left-deep tree search
10     $\mathcal{P}_l \leftarrow \emptyset$;
11     for $P_1 \in \mathcal{P}_t$ do
12         for $T \in \mathcal{T}$ do
13             $\mathcal{P}_l \leftarrow \mathcal{P}_l \cup$ {createJoin($P_1, T$)};
14     $\mathcal{P}_t \leftarrow$ findTopK($\mathcal{P}_l, k$);
       // Pruning
15     for $P_s \in \mathcal{P}_t$ do
16         if $C_{cost}(P_s, P_o) = 1$ then $\mathcal{P}_t \leftarrow \mathcal{P}_t \setminus \{P_s\}$;
17     if $|\mathcal{P}_t| = 0$ then break;
18 return findTopK($\mathcal{P}_t \cup \{P_o\}$, 1);
19 Function createJoin($P_l, P_r$)
20 return a new join node with left child $P_l$ and right child $P_r$;
21 Function findTopK($\mathcal{P}, k$)
22 return top-$k$ cheapest join plans in $\mathcal{P}$;
```

$Cost(P) = \sum_{\gamma=0}^{\Gamma} \max_{P' \in \mathcal{P}_\gamma} Card(P')$, where $\mathcal{P}_\gamma$ are the sub-plans rooted at join nodes in the γ-th layer. We adopt this cost model as it accommodates the preference for bushy tree plans in distributed systems. Then, we use $Cost(P)$ to generate labels to train C_{cost}.

Cardinality comparator C_{card}. It compares the output cardinalities of join plans P_1 and P_2 by using $Card(P)$ as $L(P)$, i.e., we use $Card(P)$ to generate labels to train C_{card}.

5 JOIN PLAN ENUMERATOR

Join Plan Enumerator generates a low-cost join plan for query Q using a join plan enumeration algorithm. This algorithm searches for left-deep tree plans progressively, and tries to generate bushy tree plans in each iteration. It then picks the cheapest join plan by pairwise comparison using C_{cost}.

Overview. Algorithm 1 outlines our join plan enumeration method. It starts with table set $\mathcal{T}$ and aims to produce a join plan composed of these tables. The algorithm initializes sub-plans $\mathcal{P}_t$ from $\mathcal{T}$'s pairwise table combinations (line 2), and iteratively extends the sub-plans in $\mathcal{P}_t$ by joining with a new table and keeps the k cheapest new sub-plans for the next iteration. When search terminates, left-deep tree plans containing all tables are generated. During each beam search iteration, the algorithm first attempts to generate a bushy tree plan that includes all tables (lines 5-9). For each sub-plan P_1, it checks whether a bushy tree plan can be generated based on P_1. If so, it searches for a join plan P_2 using the remaining tables recursively, and joins P_1 and P_2 to generate a full bushy tree plan P_b. After the bushy tree plan search, the algorithm continues to search for the left-deep tree plans (lines 10-17). As bushy tree search can obtain full plans, the algorithm maintains an optimal plan P_o, and uses its cost to prune the unpromising sub-plans. Finally, it combines the bushy tree plans and left-deep tree plans, and selects the join plan with the lowest cost. Overall, in each iteration of Algorithm 1, it first uses **bushy tree search** to generate an optimal full bushy plan based on the left-deep sub-plans generated in the previous iteration (we can generate a full bushy plan for each left-deep sub-plan and select the optimal one), and then uses **left-deep tree search** to generate k left-deep sub-plans with one additional table. We provide a running example of this in the full version [4].

Quasi-bushy tree search. In each beam search iteration, when encountering sub-plan P_1, the algorithm attempts to generate a quasi-bushy tree plan P_b containing all tables. It recursively finds join plan P_2 for the remaining tables and joins P_1 and P_2 to generate P_b. This allows P_1 and P_2 to run in parallel, enhancing performance. However, this method requires time-consuming k recursive searches per iteration. Also, bushy tree plans need to materialize intermediate tables, and the performance degrades if the outputs of P_1 or P_2 are large. Thus, we use canBeBroadcast to retain the P_1 with small output size (as detailed in Section 6), and only do the recursive searches based on such P_1. This reduces the number of recursive searches and lowers materialization costs, as P_1 join P_2 can be executed using Broadcast Join, which only materialize the small P_1. When the full plan P_b is generated, the algorithm maintains the optimal full plan P_o to prune the search space.

Left-deep tree search. After quasi-bushy tree search, the algorithm continues to search for left-deep tree plans. It starts with the sub-plans $\mathcal{P}_t$ from the previous iteration. For each sub-plan $P_1 \in \mathcal{P}_t$ (a d-table sub-plan, $2 \leq d < |\mathcal{T}|$), the algorithm joins it with a new table, generating $(d+1)$-table candidate sub-plans $\mathcal{P}_l$. It only retains the k lowest-cost sub-plans of $\mathcal{P}_l$ for the next iteration. Note that in beam search, we only consider left-deep sub-plans, since left-deep sub-plans are partial sets of tables, while the bushy plans P_o always include all tables. Thus, left-deep sub-plans consistently have lower costs than full bushy plans, so keeping the top-k plans overall is not meaningful. The search terminates when left-deep plans containing all tables are generated. Moreover, the quasi-bushy tree search yields an optimal join plan P_o. In subsequent iterations of beam search, the algorithm prunes less promising sub-plans based on P_o's cost, further reducing the search space.

Find top-k plans with comparison. Here, we outline how findTopK selects the top-k cheapest sub-plans from candidates $\mathcal{P}$. Since C_{cost} doesn't estimate exact costs, it's not straightforward to pick the k lowest-cost sub-plans directly. Thus, we employ the quick select algorithm [15], which uses pairwise comparisons to reorder sub-plans in $\mathcal{P}$ and selects the first k as the cheapest.

Comparisons with existing beam-search-based plan enumerators. Beam search is used for plan enumeration in [9, 43]. In these approaches, they use the cost of the entire query containing a specific sub-plan to guide beam search, and thus their beam search operates on search states (each a set of sub-plans for the query). However, in LEAP, beam search differs from previous approaches, since LEAP directly uses the cost of individual sub-plans to guide the beam search. Therefore, LEAP's beam search operates on single sub-plans. As a result, beam search strategies in previous

Algorithm 2: `canBeBroadcast`(P_s)

```
Input   : A sub-plan $P_s$
Output  : Whether $P_s$'s output can be broadcast
1 $t_{card} \leftarrow \frac{\text{Broadcast threshold } BJT}{\text{Estimated output width of } P_s}$;
2 $\mathcal{P}_r \leftarrow$ Set of join plans in Memory Pool with cardinality
      $\in [(1-\epsilon)t_{card}, (1+\epsilon)t_{card}]$;
3 $\mathcal{P}_h \leftarrow$ Join plans in $\mathcal{P}_r$ with top-$c$ highest uncertainty;
4 $\overline{p_{card}} \leftarrow (\sum_{P \in \mathcal{P}_h} p_{card}(P_s, P)) \,/\, |\mathcal{P}_h|$;
5 return $\overline{p_{card}} < 0.5$;
```

approaches can't be applied in Algorithm 1. LEAP additionally introduces recursive search for bushy plans and pruning techniques to better balance search efficiency and plan quality.

6 JOIN OPERATOR SELECTOR

Join Operator Selector assigns physical operators to each join operation in join plan P. First, it assesses whether the output of each sub-plan in P is suitable for broadcasting. Based on that, it assigns an appropriate physical operator to each join operation in P.

6.1 Assessing Broadcast Suitability

We discuss how to assess if a join plan's output is suitable for broadcasting. As described in Section 2.2.3, for a sub-plan P_s, Spark SQL evaluates if P_s's output can be broadcast by comparing its output size to broadcast threshold BJT. However, it's user-adjustable. Given the variability of this threshold, assessing broadcast suitability of P_s can be challenging without cardinality value estimations.

To overcome this, we use join plans with known cardinalities from the training data of C_{card} to establish a reference. We can find some join plans $\mathcal{P}_h$ with output size close to BJT, and compare P_s with $\mathcal{P}_h$ using C_{card}. Thus, we can know whether P_s's output size is smaller than the broadcast threshold BJT. If smaller, P_s's output is suitable for broadcasting. The process is outlined in Algorithm 2. For a sub-plan P_s, it first computes a cardinality threshold t_{card} based on BJT and P_s's output width estimation from Spark SQL (line 1). Then, it selects some join plans $\mathcal{P}_h$ from Memory Pool that have a cardinality near t_{card} and exhibit high uncertainty (lines 2-3). Finally, it compares P_s with each join plan in $\mathcal{P}_h$ using C_{card}, and generates an output probability p_{card} (line 4). If the average output probability $\overline{p_{card}}$ is < 0.5, it indicates P_s's output size is smaller than that of $\mathcal{P}_h$, and thus smaller than BJT. P_s's output is then considered suitable for broadcasting.

Candidate selection. In Algorithm 2, selecting candidate join plans $\mathcal{P}_h$ is critical. Here, we first identify historical plans in Memory Pool with cardinalities that fall within $[(1-\epsilon)t_{card}, (1+\epsilon)t_{card}]$, and then retain c join plans with the highest uncertainty as candidate plans $\mathcal{P}_h$. We select c join plans with cardinality near t_{card} for comparison to avoid the potential inaccuracy of a single comparison. This strategy does not harm the performance, as the default value of BJT is not always optimal. Even if P_s's output size is slightly higher than BJT (for example, within $2 \times BJT$), it can still be considered suitable for broadcasting. We prioritize candidate join plans with high uncertainty since they help avoid overly confident incorrect comparisons. Uncertainty is assessed by the L^2 distance between the representations of two join plan trees; a smaller distance indicates higher classification difficulty and uncertainty. The intuition is that, when P_1 and P_2's embedding vectors have a small L^2 distance, $Emb(P_1) - Emb(P_2)$ approaches 0 in Equation 2. After regularization, the bias term $\mathbf{b}$ is also near 0, so $\mathbf{W}(Emb(P_1) - Emb(P_2)) + \mathbf{b}$ is near 0, and the output probability $p_{card} = \sigma(\mathbf{W}(Emb(P_1) - Emb(P_2)) + \mathbf{b})$ tends toward 0.5. It reduces the impact of single incorrect comparisons on $\overline{p_{card}}$.

6.2 Assigning Physical Operator

We assign appropriate physical operators by checking if the output of each sub-plan P_s in P can be broadcast. For each join operation, we use Algorithm 2 to assess whether the output of either child sub-plan can be broadcast. If so, a BROADCAST hint is added to that join, prompting Spark SQL to execute it using Broadcast Join.

7 EXPERIMENTS

In this section, we conduct extensive experiments to validate the effectiveness of our learned Spark SQL query optimizer LEAP.

7.1 Experiment Setup

7.1.1 *Environment.* We conduct the experiments in a 3-node cluster with Spark 3.3.0. Each node is interconnected via a 1Gbps network. We assign 36 executor cores and 60GB executor memory for each query execution. We use a NVIDIA RTX-3090 GPU for model training, and use CPU for model inference during optimization.

7.1.2 *Benchmarks.* We select three typical benchmarks for evaluation by following previous studies [10, 49].

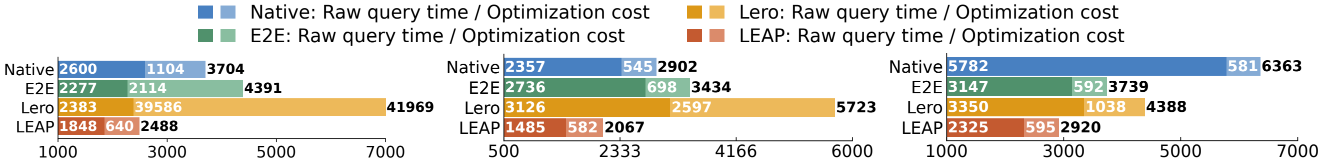
JOB [18]. Join Order Benchmark (JOB) utilizes the real-world IMDB dataset, which includes 21 tables on movies and actors. JOB comprises 113 realistic queries derived from 33 templates, with each query involving between 4 and 17 relations. We expand the IMDB tables tenfold, resulting in a 37GB dataset.

STACK [22]. STACK includes 10 tables about the information from Stack Exchange websites. The benchmark contains queries from 16 templates, with each query involving between 4 and 12 relations. The entire dataset is 100GB. In our evaluation, we exclude templates #4, 7, 9 and 10 because the first two involve arithmetic operations in join conditions, and the last two have predicate subqueries not yet supported by LEAP and all previous related works. We plan to address these limitations in future works.

TPC-H [32]. TPC-H includes 8 tables about the decision support applications in business intelligence. The benchmark contains 22 templates, and we can generate different queries based on them. We produce 100GB data as the dataset. In our evaluation, we only consider the query templates #2, 3, 5, 7, 8, 9, 10 following [49]. We exclude other templates, as they are too simple (only have one or two tables), or contain predicate subqueries.

7.1.3 *Baselines.* As many competitors can't fully optimize Spark SQL queries, we compare LEAP with three typical methods.

Spark SQL native optimizer employs Cost-Based Optimization (CBO) to estimate the output size of each join plan, and uses these estimations to guide the join plan enumeration and physical operator selection. In CBO module, it uses NDV and histograms for estimation, and we set the number of histogram bins to 64.



(a) JOB (b) STACK (c) TPC-H

Figure 5: Query performance on three benchmarks, in seconds. Black numbers outside bars represent end-to-end query execution time, while white numbers in dark and light bars represent raw query time and optimization cost respectively.

Lero [49] is a learned optimizer also using a learned comparator to choose join plans. We adopt the implementation from authors [2]. **E2E** [30] is a learned query-driven cardinality estimator based on TLSTM model. We implement it on Spark SQL based on its code [1], and replace CBO's cardinality estimation with E2E's estimation. We exclude other query-driven estimators as they lack support for disjunction [17, 19], complex string predicates [33], or bushy tree plan estimation [24]. We do not consider data-driven cardinality estimators due to their higher inference times according to [33].

We don't consider another learning-to-rank optimizer LEON [10] as it can't optimize the physical operator selection in Spark SQL. The performance drops in this case according to Section 7.4.1. We don't consider learned Spark SQL cost models [20, 21], because they only estimate the cost of PhysicalPlans in Spark SQL, not LogicalPlans. As discussed in Section 2.2.2, the optimal join order is determined by the cost of LogicalPlan, and LEAP also operates on LogicalPlan, so these models cannot be used for join order optimization. Additionally, Spark SQL's native optimizer (Catalyst) does not provide alternative PhysicalPlans for comparison in current implementation [7], making it impossible to select the best plan using a learned cost model without altering the source code.

7.1.4 Settings. We first generate training and test queries as [22, 49], and evaluate the optimizers in a static manner.

Training data generation. For each benchmark, following [22, 49], we generate 1,000 training queries based on its query templates. We randomly choose a query template, retrieve its join conditions, and add some random predicates to generate each training query. To support Lero, we use its plan exploration strategy to generate multiple candidate join plans for each training query. Thus, there are 7,600 distinct plans for JOB, 5,409 distinct plans for STACK and 3,494 distinct plans for TPC-H. This strategy is reasonable, as nearly-identical queries are frequently repeated in some OLAP workloads [22, 41]. For the test set, we employ different strategies. In JOB, we use the 113 realistic queries as test set. In STACK, we randomly select 10 queries from each query template as test set. In TPC-H, we randomly generate 10 queries for each query template.

Evaluation scenarios. All methods are evaluated in a static manner. We first train the optimizers with all training queries until convergence, then use the optimizers to run test queries and report their performance. Thus, we can compare the performance of different methods once they have been stabilized on a workload.

Evaluation metrics. We consider three key metrics in our evaluation. (1) End-to-end query execution time. This is the time a query takes from start to finish to return results. In the following text, we may also call it "query performance". (2) Optimization cost. This is the time required to generate a final join plan for query execution. (3) Raw query time. This is the duration of Spark SQL jobs associated with the query. Simply, end-to-end query execution time is the sum of raw query time and optimization cost.

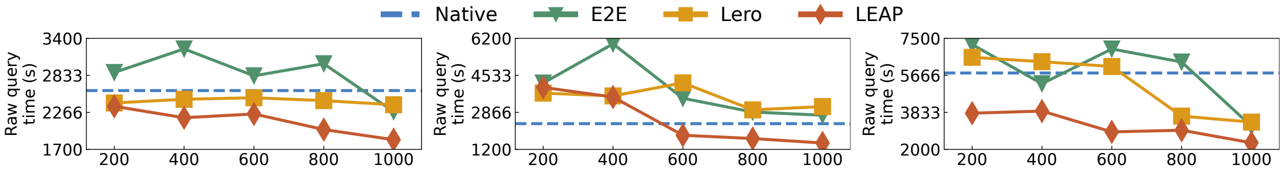
7.2 Query Performance

7.2.1 Total execution time. Figure 5 presents the total execution time for all test queries for each method. We set a timeout for raw query time of 300s to reduce the effect of long-running join plans.

End-to-end query execution time. As shown in Figure 5, LEAP outperforms the three baselines in end-to-end query execution time (the sum of dark and light bars in Figure 5). Compared with the native optimizer, LEAP reduces execution time by 32.8%, 28.8%, 54% in JOB, STACK and TPC-H respectively. Compared with E2E, LEAP reduces execution time by 43.3%, 39.8%, 21.9% in JOB, STACK and TPC-H respectively. These results showcase the effectiveness of our learning-to-rank approach over regression-based methods. Compared with Lero, LEAP reduces execution time by 94.1% in JOB, 63.9% in STACK, and 33.5% in TPC-H. Lero's much longer execution time is due to its high optimization cost, as analyzed below.

Optimization cost. We compare the optimization cost (the light bars in Figure 5) of all methods mentioned above. Compared with the native optimizer and E2E, LEAP reduces optimization cost by 42% and 69.7% respectively in JOB. Native optimizer and E2E experience faster growth in the search space for queries with more tables, while LEAP's search space maintains stable growth, as reported in Section 7.3.2. While in STACK and TPC-H, LEAP shows similar optimization costs due to the smaller search space with fewer tables and possible join conditions. Compared with Lero, LEAP reduces the optimization cost by 98.4%, 77.6%, 42.7% in JOB, STACK and TPC-H respectively. Lero incurs high optimization costs because it generates $|\alpha| \cdot |\mathcal{T}(Q)|$ candidate join plans with the native optimizer for each query Q (α is the set of scaling factors), resulting in an optimization cost several times higher than that of native optimizer.

Raw query time. We compare the raw query time without optimization cost (the dark bars in Figure 5) to demonstrate LEAP's effectiveness in finding good join plans. Compared with native optimizer, LEAP reduces raw query time by 29%, 37%, 60% in JOB, STACK and TPC-H respectively. Compared with E2E, LEAP reduces raw query time by 19%, 46%, 26% respectively, as E2E still suffers from high cardinality estimation errors, while our approach reduces such errors by an estimation-free pairwise comparison. Compared with Lero, LEAP reduces the raw query time by 23%, 53%, 31% respectively, since LEAP's Learned Comparator integrates predicate information and data features to make more accurate comparisons, instead of relying on Spark SQL CBO's inaccurate estimation. Interestingly, we find that comparison-based Lero performs worse than



(a) JOB, raw query time (b) STACK, raw query time (c) TPC-H, raw query time

Figure 6: Query performance with varying numbers of training queries of different learning-based methods.

Table 1: Total raw query time of E2E using native cardinality estimates to replace predicate information, in seconds.

	JOB	STACK	TPC-H
E2E	2,277.0	2,736.0	3,147.0
E2E with native estimates	2,744.7	3,910.6	6,282.0
Lero	2,383.0	3,126.0	3,350.0

Table 2: The speedup ratio of Balsa and LEAP over native optimizer in training and test queries. The higher, the better.

	JOB (RandSplit)		JOB (SlowSplit)		STACK		TPC-H	
	Train	Test	Train	Test	Train	Test	Train	Test
Balsa	0.75	0.58	0.67	0.70	0.35	0.39	2.00	0.89
LEAP	**1.39**	**1.47**	**1.39**	**1.47**	**1.61**	**1.38**	**2.74**	**1.00**

regression-based E2E in raw query time. First, Lero may generate more sub-optimal candidate plans with its plan exploration strategy in Spark SQL. By scaling up the estimated sub-query cardinality, it may convert Broadcast Hash Join to Sort Merge Join, leading to large table shuffles, and conversely, it may trigger large table broadcasts. Both cases will incur huge network transfer costs, causing the execution time to grow faster. Thus, Lero is more likely to choose a sub-optimal plan due to the large number of sub-optimal candidates. Second, E2E uses predicate information and sample bitmaps to capture fine-grained details of join conditions and filters, which Lero does not support. We evaluate E2E's raw query time using only the native cardinality estimates, as shown in Table 1, and it shows that E2E's raw query time is higher than that of Lero with only native cardinality estimates.

7.2.2 *Comparison with Balsa.* We evaluate the speedup ratio of Balsa and LEAP compared to the native optimizer in terms of total raw query time, as shown in Table 2. Across four benchmarks, LEAP shows a significantly higher speedup ratio than Balsa for both training and test queries. There are three main reasons for this. First, Balsa relies on a regression model to predict query latency during plan enumeration, which often leads to the selection of inappropriate physical operators due to inaccuracies in the model. Also, Balsa's on-policy learning approach struggles with limited training plans, as it can't efficiently utilize the information from earlier iterations. Lastly, Balsa's beam search is limited by expanding only one state per step. In contrast, LEAP expands all candidate sub-plans at each step, enabling a broader search for better plans.

7.2.3 *Total execution time in other systems.* We evaluate LEAP in two additional big data query processing systems, Presto [29] and Apache Doris [3], and compare LEAP's performance with their native cost-based optimizers in Table 3. We select these two systems as they support cost-based exact join order enumeration. Although two systems use different cost estimation strategies, LEAP still outperforms them by up to 29.4% in Presto and 56.4% in Apache Doris, since it provides more accurate estimations via learned approach. Note that LEAP underperforms Apache Doris native optimizer in STACK since LEAP takes too much time to rewrite the queries, and we will try to fix this issue in our future works.

Table 3: Total end-to-end query execution time of LEAP in Presto and Apache Doris, in seconds.

	JOB	STACK	TPC-H
Presto (native)	2,649.4	1,845.7	2,785.6
Presto (with LEAP)	**1,870.6**	**1,404.5**	**2,463.9**
Apache Doris (native)	1,165.9	**271.9**	950.6
Apache Doris (with LEAP)	**508.4**	325.3	**783.8**

7.2.4 *Training efficiency.* We compare the query performance of learning-based methods with varying numbers of training queries. This evaluation simulates the scenario where the model is continuously updated during workload execution, as described in [49]. The raw query times are shown in Figure 6, and we can observe that, (1) LEAP consistently outperforms E2E and Lero. For example, in JOB, LEAP reduces raw query time by up to 33% compared with E2E, and 22% compared with Lero. (2) With more training data, raw query time generally decreases as models can more accurately capture the relations between cost and plans. Notably, with more training queries, the performance gap between LEAP and baselines shrinks in TPC-H, and E2E and Lero exhibit unstable performance in STACK. This is due to the complexity of TPC-H and STACK, where some join plans can produce intermediate tables with up to 10^{11} rows, leading to query timeouts. Baselines may generate such poor plans with insufficient training data. (3) LEAP can quickly adapt to new workloads. It outperforms the native optimizer with only 200 training queries in JOB, 600 in STACK and 200 in TPC-H. The improvement continues to grow with more training queries. Note that LEAP is not specifically designed for dynamic workloads, but it performs better than baselines in this case due to their own limitations. For E2E, LEAP performs better than it due to the superiority of comparison-based approach over regression-based approach. For comparison-based Lero, it tends to generate many sub-optimal candidate plans. With limited training data, the accuracy of plan comparator is affected, and thus a plan with inferior performance is easier to be selected. Generally, LEAP avoids the limitations of baselines. Also, LEAP has fewer parameters with short input plan node sequences. This smaller data volume makes overfitting less likely with limited training data. Thus, it helps LEAP generalize better to unseen queries.

Table 4: The number of queries showing performance improvement/regression compared with native optimizer.

(a) JOB

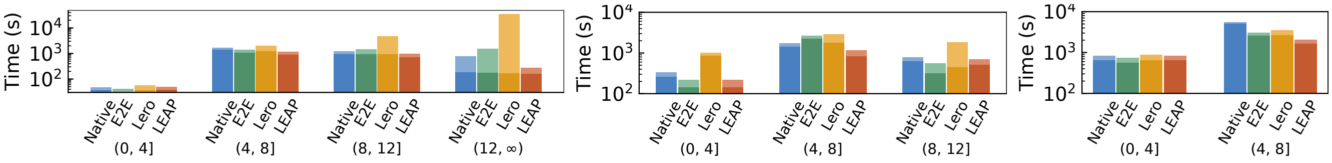
		E2E	Lero	LEAP
Improvement		65	10	96
Regression	(-20%, 0)	21	11	12
	(-∞, -20%)	25	92	5
Failure		2	0	0

(b) STACK

		E2E	Lero	LEAP
Improvement		94	9	107
Regression	(-20%, 0)	11	18	7
	(-∞, -20%)	7	88	6
Failure		8	5	0

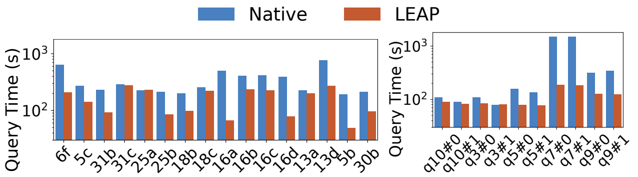
(c) TPC-H

		E2E	Lero	LEAP
Improvement		41	16	56
Regression	(-20%, 0)	4	27	14
	(-∞, -20%)	15	27	0
Failure		10	0	0



(a) JOB (b) STACK (c) TPC-H

Figure 7: Total end-to-end query execution time for queries with varying number of tables on three benchmarks. The dark and light bars represent raw query time and optimization cost respectively.



(a) JOB (b) TPC-H

Figure 8: The running time of individual large queries in augmented JOB and TPC-H, in seconds.

7.3 Query Performance on Individual Queries

7.3.1 Performance regression analysis. We compare the end-to-end query execution time of learning-based methods with Spark SQL native optimizer on each test query. For query Q executed with a learning-based method M, we compute its performance change as $\Delta(Q, M) = \frac{Time(Q,CBO)-Time(Q,M)}{Time(Q,CBO)}$, where $Time(Q, CBO)$ and $Time(Q, M)$ represent the execution time of Q with native optimizer and method M. $\Delta(Q, M) \geq 0$ indicates improvement, while $\Delta(Q, M) < 0$ indicates regression. Table 4 shows the distribution of $\Delta(Q, M)$ for each method M. LEAP achieves the least performance regression and provides significantly more performance gains compared to E2E and Lero. Although E2E matches LEAP in the number of improved queries in STACK, it has 7 long-running queries and 8 queries that fail after long execution times, leading to much higher overall query times. We find that LEAP's underperformance compared to the baselines is mainly due to incorrect physical operator selection. In STACK q8, LEAP overestimates the output cardinality of two sub-plans, resulting in shuffling two large tables and a 20% slowdown. In TPC-H q7, LEAP underestimates cardinality, leading to inefficient broadcasts of two large intermediate tables and making it 150% slower than Lero. These issues stem from the limited accuracy of the cardinality comparator C_{card}. We plan to improve this by adding more join features in future work. Conversely, in queries where LEAP excels, the improvement is due to better join order selection. For example, in STACK q15, E2E times out due to a join order with an intermediate table of 14.3 billion rows, causing excessive shuffling. LEAP, however, selects a more efficient order, producing intermediate tables of up to only 479K rows and completing the query in 15 seconds.

7.3.2 Findings on individual queries. On the level of individual queries, we can make the following findings. (1) LEAP's performance greatly improves with queries involving many tables and join conditions (e.g., JOB's 29a, 29b, 29c and STACK's q2, q3). We group the test queries of each benchmark into four categories based on the number of tables: $(0, 4]$, $(4, 8]$, $(8, 12]$, and $(12, \infty)$, and report the total end-to-end query execution time for each group in Figure 7. For queries with more tables, the larger search space makes it difficult for baselines to enumerate join plans efficiently, allowing LEAP to reduce optimization costs. For example, for queries with ≥ 8 tables, LEAP is up to 79.5% faster in JOB and 62.8% in STACK. However, with fewer tables and join conditions, the improvement is less pronounced, as the search space is smaller, making it easier for baseline methods to match LEAP's recommended execution plans, and in some cases, query time is driven more by reading or shuffling large tables than by join order (e.g., TPC-H's q3, q10). (2) LEAP's performance is related to the types of filters included in the query. Briefly, the performance improvement of LEAP is more significant in queries with predicates on skewed string columns, or pattern matching predicates (e.g., JOB's 20b, 5c, 27c). Spark SQL's estimator does not support these cases very well, and often produces inaccurate estimations, leading to inferior plans.

7.3.3 Query performance in large queries. To assess LEAP's effectiveness on larger queries, we execute some long-running queries on augmented JOB and TPC-H datasets in Figure 8. For JOB, we expand the IMDB tables to create a 400GB dataset and run the top-16 slowest queries. For TPC-H, we generate 200GB data and run two queries per template for 5 time-consuming templates. We can observe that, (1) for JOB, LEAP reduces query time by up to 74.6% compared with native optimizer. However, for some queries like 31c and 25a, LEAP achieves similar performance due to significant time spent on broadcasting the large table *cast_info*, which stems from inaccurate size estimates by the native optimizer. (2) for TPC-H, LEAP reduces the runtime of the two longest-running queries (q7 and q9) by up to 87.8% by identifying better join orders that minimize intermediate table sizes. For q10 and q3, LEAP's performance is slightly better since these queries involve fewer tables, allowing the native optimizer to find similar plans.

Table 5: Total end-to-end query execution time of LEAP with different components, in seconds. Each cell shows the end-to-end query time (e.g., 2,488.4) on top, with raw query time (e.g., 1,848) and optimization cost (e.g., 640) in brackets below.

	JOB	STACK	TPC-H
LEAP	**2,488.4** (1,848+640)	**2,067.7** (1,485+582)	**2,920.4** (2,325+595)
LEAP w/o Join Plan Enumerator	3,380.9 (2,263+1,117)	2,666.1 (2,124+541)	6,090.8 (5,490+600)
LEAP w/o Join Operator Selector	3,034.5 (2,394+640)	2,469.3 (1,895+574)	3,117.3 (2,498+618)
LEAP w/o Bushy tree search	2,607.4 (2,028+579)	2,237.9 (1,694+543)	3,363.1 (2,770+592)
LEAP with Cardinality-driven plan selection	2,671.7 (1,953+717)	2,377.0 (1,817+559)	3,298.9 (2,705+593)
LEAP with Low-uncertainty plan selection	2,843.2 (2,097+746)	2,770.3 (2,156+614)	3,798.1 (3,141+657)
LEAP with only Native estimates	4,963.8 (4,281+682)	6,283.9 (5,690+594)	4,672.6 (4,016+657)

7.3.4 Proportion of bushy plans. Here we discuss the proportion of test queries where bushy plan P_o is selected. In our experiments, 35 (31%) queries in JOB, 20 (17%) queries in STACK, and 10 (14%) queries in TPC-H adopt the bushy plan P_o. Our approach only uses sub-plans with few output rows to generate bushy plans, so the number of bushy plans is limited. Also, bushy plans are not always more efficient than left-deep plans. To improve the proportion of selected bushy plans, we can retain all generated bushy plans for the final comparison (line 18 in Algorithm 1), rather than only keeping the optimal one P_o. This prevents potentially overlooking better bushy plans, as the comparator model is not always accurate.

7.4 Ablation Study

7.4.1 Effects of each component in LEAP. In this section, we compare the performance of LEAP with some components disabled to understand the benefits of each component. We evaluate LEAP against six variants: (1) without Join Plan Enumerator, (2) without Join Operator Selector, (3) without bushy tree search in `findBestPlan`, (4) selecting candidate plans with the closest cardinality to t_{card} in Algorithm 2, (5) selecting candidate plans with lowest uncertainty to P_s in Algorithm 2, and (6) using Spark SQL's native cardinality estimates to replace predicate information and data features in Learned Comparator. The results are shown in Table 5. Without Join Plan Enumerator, performance drops by up to 108% due to the inferior join orders produced by native optimizer with large intermediate tables. Without Join Operator Selector, query execution time is up to 21.9% higher, as incorrect join operator selection causes large table shuffles or broadcasts. Without bushy tree search, performance drops by up to 15.2%, as parallel joins are disallowed. Although bushy tree search reduces raw query time by up to 16.1%, the optimization cost is higher due to larger search space. In Algorithm 2, selecting candidate plans with the closest cardinality to t_{card} or the lowest uncertainty reduces LEAP's performance by up to 34% compared to our high-uncertainty-guided candidate plan selection. High-uncertainty candidates help correct final decisions when faced with single incorrect comparisons, as they decrease the impact on average output probability $\overline{p_{card}}$. Finally, replacing features with native cardinality estimates negatively affects performance due to high estimation errors, which hinder comparator's ability to accurately compare sub-plans, particularly in Join Plan Enumerator which compares sub-plans across different queries.

Table 6: Total end-to-end query execution time of native optimizer using LEAP's either component, in seconds.

	JOB	STACK	TPC-H
Native	3,704.9 (2,600+1,104)	2,902.6 (2,357+545)	6,363.2 (5,782+581)
Native with Learned Comparator	6,455.4 (2,191+4,264)	2,892.1 (2,038+854)	3,307.1 (2,643+664)
Native with Join Plan Enumerator	3,291.2 (2,793+497)	4,859.4 (4,347+512)	9,967.7 (9,375+592)
LEAP	**2,488.4** (1,848+640)	**2,067.7** (1,485+582)	**2,920.4** (2,325+595)

Table 7: Total raw query time of LEAP using different comparator architecture and different number of training queries, in seconds.

	JOB	STACK	TPC-H
Tree convolution, 1000 queries	1,985.4	1,649.2	3,147.9
Tree convolution, 1500 queries	1,837.0	1,397.3	2,539.1
Tree-LSTM, 1000 queries	1,970.0	1,648.8	2,851.1
Tree-LSTM, 1500 queries	1,902.8	1,545.8	2,608.7
LSTM (ours), 1000 queries	1,848.0	1,485.0	2,325.0

7.4.2 Effects of learned comparator and plan enumerator in LEAP. We investigate the effects of LEAP's Learned Comparator and Join Plan Enumerator by using each of them in native optimizer. Table 6 shows that, (1) using LEAP's Learned Comparator to guide plan enumeration reduces raw query time, as it helps to find better plans with more accurate plan comparisons. However, optimization costs increase significantly because neural network inference is more time-consuming. (2) when using LEAP's Join Plan Enumerator to generate plans with traditional estimators, performance significantly declines compared with native optimizer due to large estimation errors from the traditional estimator, and our enumeration algorithm only explores a limited part of the join plan space.

7.4.3 Different comparator architecture. We further compare the performance of LEAP with different model architectures for Learned Comparator to show the efficiency of our approach. We implement our Learned Comparator using a tree convolution network, Tree-LSTM, and LSTM (our approach) and compare their raw query time, as the search space of different models are affected by C_{card}, thus their optimization cost differs. As shown in Table 7, with 1000 training queries, our LSTM-based approach outperforms the others across all three benchmarks, reducing raw query time by up to 6.9%, 10% and 26.1% in JOB, STACK and TPC-H respectively. Tree models find worse plans for some queries due to their structural limitations. Tree-LSTM may emphasize the root node, placing less emphasis on leaf nodes [36], while tree convolution networks can suffer from information dilution and ineffective feature aggregation when capturing long-distance dependencies in join plan trees [8]. Also, join plan trees are simple binary structures with limited height, which limit the benefits of tree models. In this case, tree models can increase complexity and potentially lead to overfitting with limited training data. Thus, tree models underperform in some cases compared to LSTM, which aligns with previous findings in [48]. We also train the tree models with 1500 training queries, and while their performance improves, they can't surpass the performance of LSTM with 1000 training queries in many cases.

8 RELATED WORK

In this section, we review the related works of cardinality estimation and learned query optimizers.

8.1 Cardinality estimation

Traditional estimators. Native optimizers typically use histograms, sketches, and sampling techniques for cardinality estimation. Histograms [27, 34] capture the attribute value distribution. Sketches [13, 28] represent a column as a vector or matrix to assess data statistics. Sampling methods [11, 12, 37, 39] sample tuples from multiple tables to discern inter-table correlations. However, traditional estimators fail to capture inter-table correlations using histograms or sketches, and face high variance when the sample distribution differs from the real distribution [14, 19, 31].

Learned data-driven estimators. Learned data-driven estimators [35, 40, 44, 50] employ various machine learning models to understand the joint distribution of underlying data, such as deep autoregressive model in NeuroCard [44], factorize-split-sum-product network (FSPN) in FLAT [50], and normalizing flow in FACE [35]. However, learned data-driven estimators incur high training and inference costs, which escalate as the data volume increases.

Learned query-driven estimators. Learned query-driven estimators [17, 19, 25, 30, 33, 47] address cardinality estimation as a regression problem. They collect (query, cardinality) pairs and use various machine learning models to model their relationship, such as multi-set convolutional network [17], TLSTM [30], attention-based model [19] and SRU [33]. These methods have limited applicability, especially with complex string predicates and disjunctions (i.e., 'OR'). Also, they generally offer less precise estimates than those derived from learned data-driven estimators.

8.2 Learned query optimizers

Instead of estimating the query cardinality to help generate low-cost query plans, learned query optimizers directly optimize the query plans to reduce the query costs. There are primarily plan-constructor and plan-steerer methods, and some learned-comparator-based methods have also emerged recently.

Plan-constructor. Plan-constructor methods [9, 23, 43, 46] often build a new learned optimizer that discards or underuses expert knowledge of native optimizer, and generate an execution plan from scratch for each query Q. These methods use deep reinforcement learning models [6, 38] trained on historical query data to estimate query cost, and propose various plan search algorithms to generate execution plans. Compared with such methods, LEAP designs a comparator to compare sub-plan costs instead of estimating overall query costs. Additionally, LEAP introduces a plan enumeration algorithm that separates the search for left-deep and bushy plans, along with a customized physical operator selection mechanism tailored for Spark SQL, rather than relying on operator enumeration in DBMS.

Plan-steerer. Plan-steerer methods [10, 22, 45, 49] leverage the knowledge of the native optimizer to produce improved execution plans. These methods guide the native optimizer to generate multiple candidate plans, and select the plan with the lowest estimated cost. For example, BAO [22] uses the native optimizer to create a plan for each hint set, HybridQO [45] generates candidate plans using leading hints, Lero [49] adjusts the native optimizer's cardinality estimates to produce plans, while LEON [10] enumerates sub-plans with native optimizer. While these methods adapt well to changes in data and schema, their dependency on specific DBMS features restricts their use in platforms like Spark SQL. Also, generating execution plans in Spark SQL is slower than in other DBMS, significantly raising optimization costs.

Learned comparators. Learned comparator methods [10, 42, 49] use a comparator to select the best execution plan from candidate sets. While they share a similar comparator architecture (tree convolution model, using native cardinality and cost estimation as features), they differ in how they generate candidate plans. While our LEAP and existing methods share a similar framework of a plan comparator and plan generation strategy, the components differ significantly. For the plan comparator, LEAP incorporates predicate information (e.g., join conditions and filters) and data features (e.g., histograms and selectivity distributions), avoiding inaccurate native cardinality estimates in Spark SQL. Also, LEAP uses a lightweight LSTM model with structure-preserving traversal, enabling better plan generation with limited training data. Moreover, LEAP can train using plans from different queries, while other methods require multiple plans for the same query, increasing their training overhead. As for plan generation, LEAP adopts a unique approach by constructing a single plan from scratch, bypassing the native optimizer. In contrast, existing methods use plan-steerer methods to produce multiple candidate plans with the help of native optimizer.

9 CONCLUSION

In this paper, we propose LEAP, a learned query optimizer tailored for Spark SQL. To tackle compatibility, LEAP optimizes queries by first enumerating join plans and then assigning physical operators, aligning with Spark SQL and requiring no changes for integration. Also, LEAP enhances the join plan selection with a Learned Comparator to avoid the inaccuracy of exact value estimation, and provides more accurate comparison by using predicate information and data features. Moreover, LEAP adopts a progressive join plan enumeration algorithm with a beam search strategy and pruning techniques, ensuring efficient and effective join plan generation. Extensive experiments on public benchmarks demonstrate that LEAP delivers superior performance in end-to-end execution time, optimization cost, and training efficiency.

ACKNOWLEDGMENTS

This work was supported in part by the NSFC under Grants No. (62472377, 62025206, U23A20296, 62302436), and Zhejiang Province's "Lingyan" R&D Project under Grant No 2024C01259. Lu Chen and Tianyi Li are the corresponding authors of the work.

REFERENCES

[1] 2019. https://github.com/greatji/Learning-based-cost-estimator

[2] 2023. https://github.com/Blondig/Lero-on-Spark

[3] 2023. https://github.com/apache/doris

[4] 2024. https://github.com/HuashiSCNU0303/LEAP/tree/main/full_version

[5] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. 2015. Spark sql: Relational data processing in spark. In *SIGMOD*. 1383–1394.

[6] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. 2017. Deep reinforcement learning: A brief survey. *IEEE Signal Processing Magazine* 34, 6 (2017), 26–38.

[7] Lorenzo Baldacci and Matteo Golfarelli. 2019. A Cost Model for SPARK SQL. *IEEE Transactions on Knowledge and Data Engineering* 31, 5 (2019), 819–832. https://doi.org/10.1109/TKDE.2018.2850339

[8] Baoming Chang, Amin Kamali, and Verena Kantere. 2024. A Novel Technique for Query Plan Representation Based on Graph Neural Nets. In *International Conference on Big Data Analytics and Knowledge Discovery*. Springer, 299–314.

[9] Tianyi Chen, Jun Gao, Hedui Chen, and Yaofeng Tu. 2023. LOGER: A Learned Optimizer Towards Generating Efficient and Robust Query Execution Plans. *Proc. VLDB Endow.* 16, 7 (2023), 1777–1789.

[10] Xu Chen, Haitian Chen, Zibo Liang, Shuncheng Liu, Jinghong Wang, Kai Zeng, Han Su, and Kai Zheng. 2023. LEON: A New Framework for ML-Aided Query Optimization. *Proc. VLDB Endow.* 16, 9 (2023), 2261–2273.

[11] Yu Chen and Ke Yi. 2017. Two-Level Sampling for Join Size Estimation. In *SIGMOD*. 759–774.

[12] C. Estan and J.F. Naughton. 2006. End-biased Samples for Join Cardinality Estimation. In *ICDE*. 20–20.

[13] Sumit Ganguly, Minos N. Garofalakis, and Rajeev Rastogi. 2004. Processing Data-Stream Join Aggregates Using Skimmed Sketches. In *International Conference on Extending Database Technology*. https://api.semanticscholar.org/CorpusID:11330374

[14] Jintao Gao, Zhanhuai Li, Wenjie Liu, Zhijun Guo, and Yantao Yue. 2020. A new fragments allocating method for join query in distributed database. *Frontiers of Computer Science* 14, 4 (2020), 144608.

[15] Charles AR Hoare. 1961. Algorithm 65: find. *Commun. ACM* 4, 7 (1961), 321–322.

[16] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *ICPC*. 200–210.

[17] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter A. Boncz, and Alfons Kemper. 2018. Learned Cardinalities: Estimating Correlated Joins with Deep Learning. *ArXiv* abs/1809.00677 (2018).

[18] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *Proc. VLDB Endow.* 9, 3 (2015), 204–215.

[19] Pengfei Li, Wenqing Wei, Rong Zhu, Bolin Ding, Jingren Zhou, and Hua Lu. 2023. ALECE: An Attention-based Learned Cardinality Estimator for SPJ Queries on Dynamic Workloads. *Proc. VLDB Endow.* 17, 2 (2023), 197–210.

[20] Yan Li, Liwei Wang, Sheng Wang, Yuan Sun, and Zhiyong Peng. 2022. A Resource-Aware Deep Cost Model for Big Data Query Processing. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. 885–897.

[21] Yan Li, Liwei Wang, Sheng Wang, Yuan Sun, Bolong Zheng, and Zhiyong Peng. 2024. A learned cost model for big data query processing. *Information Sciences* 670 (2024), 120650.

[22] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2021. Bao: Making Learned Query Optimization Practical. In *SIGMOD*. 1275–1288.

[23] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: a learned query optimizer. *Proc. VLDB Endow.* 12, 11 (2019), 1705–1718.

[24] Parimarjan Negi, Ryan Marcus, Andreas Kipf, Hongzi Mao, Nesime Tatbul, Tim Kraska, and Mohammad Alizadeh. 2021. Flow-loss: learning cardinality estimates that matter. *Proc. VLDB Endow.* 14, 11 (2021), 2019–2032.

[25] Parimarjan Negi, Ziniu Wu, Andreas Kipf, Nesime Tatbul, Ryan Marcus, Sam Madden, Tim Kraska, and Mohammad Alizadeh. 2023. Robust Query Driven Cardinality Estimation under Changing Workloads. *Proc. VLDB Endow.* 16, 6 (2023), 1520–1533.

[26] Arjan Pellenkoft, César A. Galindo-Legaria, and Martin L. Kersten. 1997. The Complexity of Transformation-Based Join Enumeration. In *VLDB*. 306–315.

[27] Viswanath Poosala and Yannis E. Ioannidis. 1997. Selectivity Estimation Without the Attribute Value Independence Assumption. In *VLDB*.

[28] Florin Rusu and Alin Dobra. 2008. Sketches for size of join estimation. *ACM Trans. Database Syst.* 33, 3 (2008), 46.

[29] Raghav Sethi, Martin Traverso, Dain Sundstrom, David Phillips, Wenlei Xie, Yutian Sun, Nezih Yegitbasi, Haozhun Jin, Eric Hwang, Nileema Shingte, and Christopher Berner. 2019. Presto: SQL on Everything. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. 1802–1813. https://doi.org/10.1109/ICDE.2019.00196

[30] Ji Sun and Guoliang Li. 2019. An end-to-end learning-based cost estimator. *Proc. VLDB Endow.* 13, 3 (2019), 307–319.

[31] Ji Sun, Jintao Zhang, Zhaoyan Sun, Guoliang Li, and Nan Tang. 2021. Learned cardinality estimation: a design space exploration and a comparative evaluation. *Proc. VLDB Endow.* 15, 1 (2021), 85–97.

[32] Transaction Processing Performance Council (TPC). 2021. TPC-H Version 2 and Version 3. http://www.tpc.org/tpch/

[33] Fang Wang, Xiao Yan, Man Lung Yiu, Shuai LI, Zunyao Mao, and Bo Tang. 2023. Speeding Up End-to-end Query Execution via Learning-based Progressive Cardinality Estimation. *Proc. ACM Manag. Data* 1, 1 (2023), 25.

[34] Hai Wang and Kenneth C. Sevcik. 2003. A multi-dimensional histogram for selectivity estimation and fast approximate query answering. In *Proceedings of the 2003 Conference of the Centre for Advanced Studies on Collaborative Research*. 328–342.

[35] Jiayi Wang, Chengliang Chai, Jiabin Liu, and Guoliang Li. 2021. FACE: a normalizing flow based cardinality estimator. *Proc. VLDB Endow.* 15, 1 (2021), 72–84.

[36] Jin Wang, Liang-Chih Yu, K. Robert Lai, and Xuejie Zhang. 2019. Investigating Dynamic Routing in Tree-Structured LSTM for Sentiment Analysis. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, Kentaro Inui, Jing Jiang, Vincent Ng, and Xiaojun Wan (Eds.). Association for Computational Linguistics, Hong Kong, China, 3432–3437. https://doi.org/10.18653/v1/D19-1343

[37] TaiNing Wang and Chee-Yong Chan. 2020. Improved Correlated Sampling for Join Size Estimation. In *ICDE*. 325–336.

[38] Sai Wu, Ying Li, Haoqi Zhu, Junbo Zhao, and Gang Chen. 2022. Dynamic index construction with deep reinforcement learning. *Data Science and Engineering* 7, 2 (2022), 87–101.

[39] Wentao Wu, Jeffrey F. Naughton, and Harneet Singh. 2016. Sampling-Based Query Re-Optimization. In *SIGMOD*. 1721–1736.

[40] Ziniu Wu, Parimarjan Negi, Mohammad Alizadeh, Tim Kraska, and Samuel Madden. 2023. FactorJoin: A New Cardinality Estimation Framework for Join Queries. *Proc. ACM Manag. Data* 1, 1, Article 41 (2023), 27 pages.

[41] Jinhan Xin, Kai Hwang, and Zhibin Yu. 2022. LOCAT: Low-Overhead Online Configuration Auto-Tuning of Spark SQL Applications. In *SIGMOD*. 674–684.

[42] Xianghong Xu, Zhibing Zhao, Tieying Zhang, Rong Kang, Luming Sun, and Jianjun Chen. 2023. COOOL: A Learning-To-Rank Approach for SQL Hint Recommendations. *arXiv preprint arXiv:2304.04407* (2023).

[43] Zongheng Yang, Wei-Lin Chiang, Sifei Luan, Gautam Mittal, Michael Luo, and Ion Stoica. 2022. Balsa: Learning a Query Optimizer Without Expert Demonstrations. In *SIGMOD*. 931–944.

[44] Zongheng Yang, Amog Kamsetty, Sifei Luan, Eric Liang, Yan Duan, Xi Chen, and Ion Stoica. 2020. NeuroCard: one cardinality estimator for all tables. 14, 1 (2020), 61–73.

[45] Xiang Yu, Chengliang Chai, Guoliang Li, and Jiabin Liu. 2022. Cost-Based or Learning-Based? A Hybrid Query Optimizer for Query Plan Selection. *Proc. VLDB Endow.* 15, 13 (2022), 3924–3936.

[46] Yuchen Yuan, Xiaoyue Feng, Bo Zhang, Pengyi Zhang, and Jie Song. 2024. JAPO: learning join and pushdown order for cloud-native join optimization. *Frontiers of Computer Science* 18, 6 (2024), 186614.

[47] Yue Zhao, Gao Cong, Jiachen Shi, and Chunyan Miao. 2022. QueryFormer: a tree transformer model for query plan representation. *Proc. VLDB Endow.* 15, 8 (2022), 1658–1670.

[48] Yue Zhao, Zhaodonghui Li, and Gao Cong. 2024. A Comparative Study and Component Analysis of Query Plan Representation Techniques in ML4DB Studies. *Proc. VLDB Endow.* 17, 4 (mar 2024), 823–835. https://doi.org/10.14778/3636218.3636235

[49] Rong Zhu, Wei Chen, Bolin Ding, Xingguang Chen, Andreas Pfadler, Ziniu Wu, and Jingren Zhou. 2023. Lero: A Learning-to-Rank Query Optimizer. *Proc. VLDB Endow.* 16, 6 (2023), 1466–1479.

[50] Rong Zhu, Ziniu Wu, Yuxing Han, Kai Zeng, Andreas Pfadler, Zhengping Qian, Jingren Zhou, and Bin Cui. 2021. FLAT: fast, lightweight and accurate method for cardinality estimation. 14, 9 (2021), 1489–1502.