
# LIO: A lightweight and interpretable query optimizer based on an evolutionary forest

Chen Ye
Hangzhou Dianzi University
chenye@hdu.edu.cn

Shujie Ma
Hangzhou Dianzi University
222050243@hdu.edu.cn

Guojun Dai
Hangzhou Dianzi University
daigj@hdu.edu.cn

Hengtong Zhang
Harbin Institute of Technology
hengtong@hit.edu.cn

## ABSTRACT

Learning-based query optimizers have shown significant advantages in generating high-quality query plans. In these optimizers, query plans are represented at different level of granularity, and learning-based models are used to learn the relationship between query plans and execution times based on the past experience. Thus, efficient query plans can be generated for given queries. However, these optimizers often struggle to achieve a balance between model efficiency and prediction accuracy. In this paper, we propose a lightweight and interpretable query optimizer LIO based on an evolutionary forest. LIO employs a genetic programming algorithm to automatically explore optimal feature combinations for a random forest, balancing model usage costs, prediction accuracy, and interpretability. The outputs of the evolutionary forest serve as interpretability aids, guiding users in dynamically adding enhanced hint sets, which in turn improves optimization performance. Additionally, two pruning strategies are developed to reduce both the number and depth of the trees in the forest, significantly enhancing rule interpretability while maintaining an acceptable level of performance loss. Extensive experiments validate that LIO outperforms state-of-the-art optimizers in terms of prediction accuracy, total runtime, and interpretability.

**PVLDB Reference Format:**
Chen Ye, Shujie Ma, Guojun Dai, and Hengtong Zhang. LIO: A lightweight and interpretable query optimizer based on an evolutionary forest. PVLDB, 19(6): 1088 - 1100, 2026.
doi:10.14778/3797919.3797920
**PVLDB Artifact Availability:**
The source code, data, and/or other artifacts have been made available at https://github.com/MaxBlack214/LIO.

## 1 INTRODUCTION

Learning-based query optimizers have demonstrated significant advantages in enhancing optimization performance, particularly due to their ability to generate effective query plans for complex queries. In these optimizers, physical query plans are encoded into vectors, and learning-based models are employed to understand the inherent relationship between the plan representation and execution times based on past experiences. This allows for the generation of efficient plans for specific queries.

Encoding the relationships within a query plan can be achieved at different levels of granularity, including parent-child dependencies [13, 14, 31, 32] and long-distance dependencies [2, 30]. The former methods focus on modeling the relationships between parent nodes and their child nodes. Although these approaches are straightforward, they typically account for fixed and short dependencies, which can lead to lower prediction accuracy. In contrast, methods that model long-distance dependencies consider relationships between all nodes in the query plan, capturing information flow across long distances. However, these approaches result in highly-complex learned representations of the plans, require large amounts of training data, and lead to longer training and inference times. Consequently, they can be challenging to adapt to unseen query scenarios. In summary, these methods struggle to achieve a balance between model efficiency and prediction accuracy.

To address this drawback, we propose using a random forest model, which is lightweight and interpretable. We automatically generate combined features that captures the relationship among different nodes. This approach ensures that the captured dependencies are neither limited to only parent-child relationships nor excessively broad to include all long-distance connections, thus balancing model efficiency, prediction accuracy, and interpretability.

**Motivating Example.** Figure 1 compares the TCNN-based method Bao [13], the transformer-based method QueryFormer [30], with our proposed method, LIO. While presented for a single query for clarity, this example reflects the typical performance trends observed across our datasets, as validated in Section 4. The Bao method employs a tree-CNN model with triangular-shape filters (parent, left-child, right-child) that slides over a query plan tree, capturing the parent-child dependencies within a physical plan tree. For a given a query plan, Bao achieves a low inference time of 0.01s and a training time of 14s. However, there is a discrepancy between the predicted execution time of 3s and the actual execution time of 2s. On the other hand, QueryFormer utilizes a Transformer model with a tree-structured architecture designed to learn representations of physical query plans. In this case, QueryFormer predicts the execution time to be 12s, which is far from the actual execution time of 2s. Also, it has a longer model inference time of 3s and a training time of 45s, which results in higher deployment costs. In contrast, LIO uses random forest with combined features of different nodes

Guojun Dai and Hengtong Zhang are the corresponding authors.

Proceedings of the VLDB Endowment, Vol. 19, No. 6 ISSN 2150-8097.
doi:10.14778/3797919.3797920

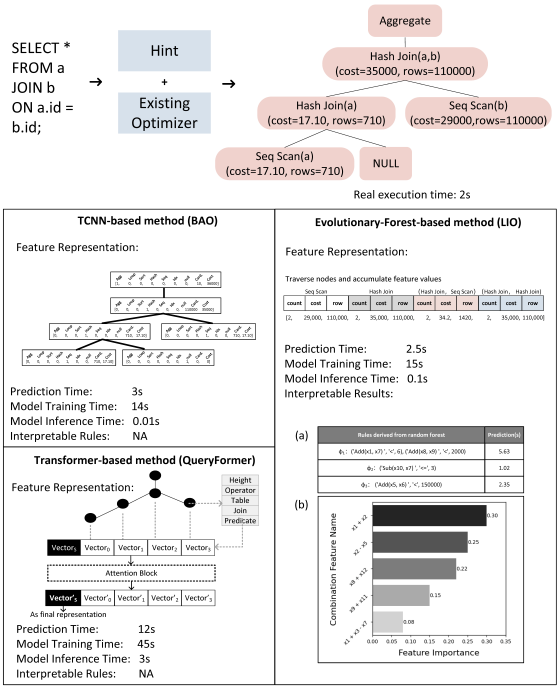



**Figure 1: A representative example comparing plan encoding and performance across query optimization methods.**

to achieve high prediction accuracy. It estimates the execution time at just 2.5s, with a training time of 15s and an inference time of only 0.1s. As shown in Figure 1(a)(b), LIO can also generate interpretable information related to query operators. This capability allows for a deeper analysis of the relationships between operator behavior and execution plans, ultimately enhancing the query optimization efficiency of LIO through adaptable hint adjustments.

**Challenges.** Designing a learned and interpretable optimizer using random forest (RF) techniques involves addressing three key challenges: (1) How to automatically select feature granularity during extraction to balance the efficiency and performance? (2) How to effectively utilize the model's outputs to enhance interpretability and, in turn, improve its query optimization capabilities? (3) How to make the random forest more compact to enhance the interpretability of the outputs and reduce inference time?

In this study, to address the first challenge, we employ a genetic programming algorithm to generate a random forest, known as an evolutionary forest (EF). This allows the evolutionary process to automatically explore optimal feature combinations and tree structures. As a result, the model can create nonlinear separating surfaces during training, thereby enhancing prediction accuracy. For the second challenge, we calculate operator weights based on the feature importance derived from the evolutionary forest. This helps us identify key operators that significantly impact query plan performance. The insights gained from this analysis enable us to update hint sets, thereby facilitating the discovery of high-quality query plans. To overcome the third challenge, we implement a clustering algorithm to reduce the number of trees in the model and an optimization framework to decrease tree depth. This approach leads to more concise output rules. Despite the pruning, the model retains its ability to identify optimized plans while significantly reducing inference time.

**Contributions.** We summarize our contributions as below.

(1) We introduce LIO, a lightweight and interpretable query optimizer that utilizes the random forest. By using a genetic programming algorithm, we generate the evolutionary forest where the evolutionary process adaptively explores optimal feature combinations and tree structures.

(2) We leverage the feature importance extracted from the evolutionary forest to guide the dynamic update of hint sets, enabling the generation of more potential high-quality query plans.

(3) To reduce inference latency and improve model interpretability, we adopt pruning strategies to decrease both the number and depth of decision trees in the evolutionary forest, balancing model complexity and prediction accuracy.

(4) Extensive experiments demonstrate that LIO achieves higher query efficiency and provides more interpretable results compared to state-of-the-art learning-based query optimizers.

## 2 PROBLEM DEFINITION

In this section, we first define some important concepts and then give the formal definition of interpretable query optimization.

**Query plan generation**. Traditional query optimizers usually transform each query $q_i$ into $n$ physical query execution plans $\mathcal{P}_i = \{p_1, \ldots, p_n\}$ according to predicate reordering, join operator ordering, and the selection of execution strategies for different types of operators. Generally, each query plan $p_i \in \mathcal{P}_i$ is modeled as a tree, where each leaf node is a scan operator *Scan* from scan set $S = \{$*Sequence Scan*, *Index Scan*, *Index Only Scan*, *Bitmap Index Scan*, $\ldots\}$, and each non-leaf node is a join operator *Join* from join set $J = \{$*Nested Loop Join*, *Hash Join*, *Merge Join*, $\ldots\}$. For the operator of each node, traditional query optimizers also return the statistics related to the operator, including cost, rows (i.e., estimated cardinality), and buff based on the internal database.

**Query hints**. Given $q_i$, a hint set $H_i$ is a set of operators to restrict the traditional optimizers from using entire operators to generate a query plan $p_i$. Specifically, each hint set $H_i$ contains at least two scan operators from $S$ and one join operator from $J$: $H_i = \{\{Scan\} \mid Scan \in S\} \cup \{\{Join\} \mid Join \in J\}$, where $|\{Scan\}| > 1$, $|\{Join\}| > 0$. Given $H_i$, the traditional optimizer will generate a tree-structured query plan $p_i$ where each node is an operator from $H_i$: $H_i : q_i \rightarrow p_i$. Then, $\mathcal{P}_i = \{p_1, \ldots, p_n\}$ is obtained based on a family of $n$ hint sets $\mathcal{H}_i = \{H_i^{(1)}, \ldots, H_i^{(n)}\}$, i.e., $\mathcal{P}_i = \{H_i(f) \mid H_i \in \mathcal{H}_i\}$.

**Interpretable Query Optimization.** Given a database $\mathcal{D}$, a workload $\mathcal{Q} = \{q_1, \ldots, q_n\}$, and an initial family of hint sets $\mathcal{H}_i$ for each $q_i \in \mathcal{Q}$, the problem of *interpretable query optimization* is two-fold:

(1) **Prediction:** For each query $q_i$, predict its optimal execution plan $p_i^* \in \mathcal{P}_i$ from the space of valid plans induced by $\mathcal{H}_i$;

(2) **Explanation:** Provide actionable and human-understandable justifications for the prediction, such as feature importance, decision rules, or rule-based interventions, so that users or the system itself can refine $\mathcal{H}_i$ with higher-quality hints in a transparent and iterative manner.

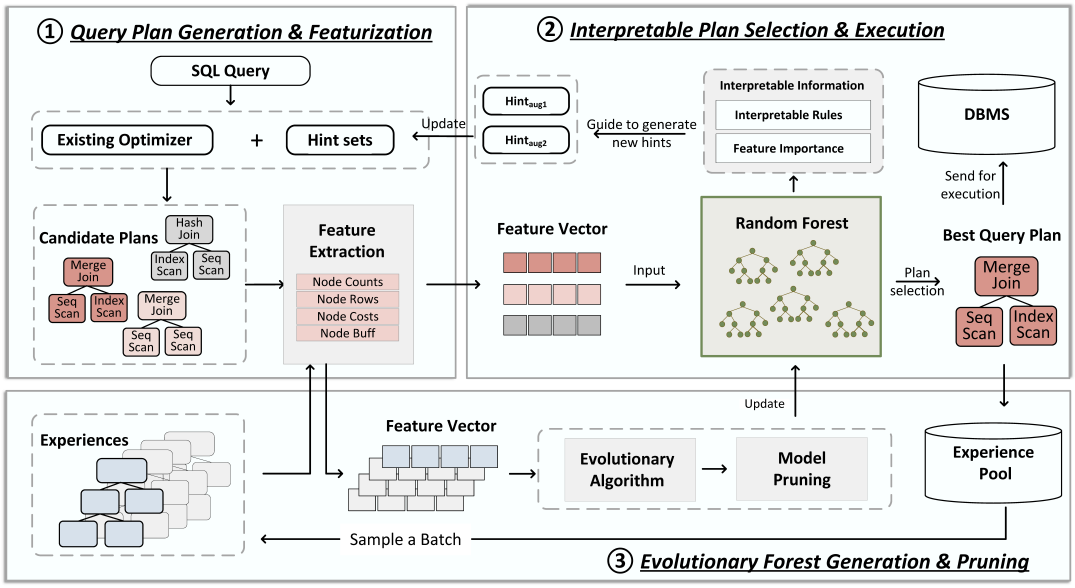



**Figure 2: LIO: A interpretable query optimizer based on EF.**

Remark: This definition emphasizes **model transparency** and **actionability**, which are key pillars of interpretability in high-stakes systems [5, 21]. Unlike purely perceptual measures (e.g., user studies), we ground interpretability in *system-level outcomes*: Whether feature importance guides effective hint refinement (Section 4.6), and whether model simplification (e.g., pruning) preserves performance while improving readability (Section 4.5). These experiments provide objective, reproducible evidence of interpretability.

## 3 THE LIO MODEL

In this section, we first give the overview of LIO and then detail each key module of LIO.

### 3.1 Framework Overview

An illustration of the LIO framework is shown in Figure 2. At a high level, LIO consists of three key modules.

*1. Query plan generation and featurization.* For an incoming query, LIO first uses an existing optimizer to generate candidate query plans. To limit the search space during plan generation, possible execution strategies are also given as hint sets. Specifically, each hint set includes a set of operations of an execution strategy. Multiple candidate query plans are then enumerated based on the given hint sets, where each node in a tree-like plan includes both the operation and its statistics returned by the database. By traversing a tree-like query plan, LIO encodes the features of each plan into a one-dimensional feature vector, where each position in the vector represents the sum of statistical information for one type of nodes.

*2. Interpretable plan selection and execution.* Given the encoded feature of a query plan, LIO designs a EF for cost prediction. In the forest, the nodes of each DT are complex features planned by a evolutionary algorithm. Each DT predicts the execution time of the given plan based on its feature vector, and the final predicted value is the average of the predictions made by all DTs in the RF. For a query, the query plan with the lowest cost among all candidate plans is selected as the best query plan for execution. The SQL result is then sent to users and the execution time is saved in an experience pool for EF training. Moreover, the feature importance from EF is used for high-quality hint set selection, so that the performance of LIO can be improved by generating better query plans with faster execution speed based on these new-added high-quality hint sets.

*3. EF generation and pruning.* Given the feature vectors of query plans, the evolutionary algorithm combines the simple features from the feature vectors into complex features for RF training and updating. Specifically, the best decision tree selected by the evolutionary algorithm is added to the archive for updating. In the archive, the tree-count and tree-depth pruning strategies are designed to reduce the inference latency. Benefiting from the update mechanism, RF is able to adapt to the current workload and produce the latest feature importance and decision rules as interpretable information for prediction and hint set update.

### 3.2 Query Plan Featurization

Based on the given hint sets, multiple query plans are generated for one query. We then present how to extract the features of each query plan and encode the extracted features into vector representation.

Existing methods typically extract features from each query plan in terms of both tree structure and node semantics. However, learning the internal relationships among these structural and semantic features often results in models that are complex and time-consuming to train. To address this issue, we propose generating one-dimensional feature vectors that encode both structural and semantic information by summarizing the statistics of different types of nodes. In addition to commonly incorporated features such as costs, estimated cardinality (rows), and buffer sizes, we also

include the number of operators (counts) in the plan as a feature. This inclusion is crucial because the number of operators directly reflects the complexity of the query. A higher number of operators often indicates a more intricate query that may require additional resources and time to execute. Understanding the distribution of operators enables the optimizer to make informed decisions about execution strategies, as different operators can significantly impact performance. Furthermore, this feature enhances the model's interpretability, allowing users to understand how the complexity of the query influences optimization decisions.

First, we divide the nodes into single-type node $s_i$ and parent-child-type node $(p_j, c_k)$. $s_i$ refers to the type of operator of each node where $s_i \in J \cup S$. $(p_j, c_k)$ denotes the double-layer node that has a parent-child relationship in the query plan tree, which is used to reserve the structural dependency among parent node $p_j \in J$ and child node $c_k \in J \cup S$ of the tree. Specifically, if both the parent node and child node are *Join* operations, it is a (*Join, Join*) type. If the parent node is a *Join* type and the child node is a *Scan* operation, it is a (*Join, Scan*) type. For instance, if the parent node is a *Hash Join* operator and the child node is a *Sequence Scan* operator, it is a (*Hash Join, Sequence Scan*) type.

Then, a query plan is traversed to identify the node types $s_i$ and $(p_j, c_k)$. To encode these nodes, we summarize the statistics returned by the database in separate ways. For each $s_i$, we encode its features into a 4-dimensional vector $\mathbf{v}_{s_i}$ by summing the counts, costs, rows, and buffer metrics of all nodes belonging to $s_i$ in the plan tree, i.e., $\mathbf{v}_{s_i} = (\sum \text{Count}_{s_i}, \sum \text{Cost}_{s_i}, \sum \text{Row}_{s_i}, \sum \text{Buff}_{s_i})$. For each $(p_j, c_k)$, we encode its features into a 4-dimensional vector $\mathbf{v}_{(P_j, C_k)}$ by summarized its counts, costs, rows, and buff. Specifically, the count is incremented by 1, while the costs, rows, and buff metrics are averaged between the two nodes $P_j$ and $C_k$, i.e., $\mathbf{v}_{(P_j,C_k)} = (\sum \text{Count}_{(p_j,c_k)}, \sum \frac{\text{Cost}_{p_j}+\text{Cost}_{c_k}}{2}, \sum \frac{\text{Row}_{p_j}+\text{Row}_{c_k}}{2}, \sum \frac{\text{Buff}_{p_j}+\text{Buff}_{c_k}}{2})$.

Finally, the feature vector $\mathbf{v}$ of a query plan $p$ is obtained by concatenating all the vectors $\mathbf{v}_{s_i}$ and $\mathbf{v}_{(p_j,c_k)}$ of nodes in its query plan tree, i.e., $\mathbf{v} = \mathbf{v}_{s_1} \oplus \mathbf{v}_{s_2} \oplus \cdots \oplus \mathbf{v}_{s_i} \oplus \mathbf{v}_{(p_1,c_1)} \oplus \mathbf{v}_{(p_1,c_2)} \oplus \cdots \oplus \mathbf{v}_{(p_j,c_k)}$. Let $m$ denote the total number of distinct operator types, including $s_i$ and $(p_j, c_k)$. Once the operator vocabulary is fixed during system setup, $m$ becomes a constant, and thus the dimension of the feature vector $\mathbf{v}$ is fixed at $1 \times 4m$. To make the description more clear, we denote the vector $\mathbf{v}$ as $\mathbf{v} = [x_1, x_2, \ldots, x_n], n = 4m$. By encoding the plan to $\mathbf{v}$, each feature in $\mathbf{v}$ has a clear physical meaning, providing effective interpretability.

**Example 1.** Consider the query plan shown in Figure 1, LIO encodes the plan into a feature vector that includes statistics information from two types of nodes: single-type nodes (*Seq Scan and Hash Join*) and parent-child-type nodes ((*Hash Join, Seq Scan*) and (*Hash Join, Hash Join*)). The features $x_1$, $x_2$, and $x_3$ are derived by calculating the total count, total cost, and total rows of the *Seq Scan* nodes in the query plan. Specifically, the total count is 2, the total cost is 29,000, and the total number of rows is 110,000, respectively. These statistics are derived from the query plan's execution estimates provided by PostgreSQL's `EXPLAIN` output.

## 3.3 Interpretable Plan Selection and Execution

In this section, we first discuss the motivation for utilizing EF. Then, we propose how to use EF for query plan selection and execution. Finally, we describe how to update the hint sets based on the feature importance derived from EF.

*3.3.1 Advantages of EF..* Database queries need timely responses. If the training time of a model is excessively long and the efficiency gains from optimization do not compensate for this training overhead, overall effectiveness may decline. Compared to existing learning-based methods, the RF model provides interpretable insights, requires less training time, and incurs lower computational costs, making it a more advantageous option for optimization tasks. Additionally, we use genetic programming to create feature combinations that capture long-range dependencies between the elements of $\mathbf{v}$. This approach replace individual features in $\mathbf{v}$ with feature constraints for DT splits, resulting in a more efficient generation of the EF model.

*3.3.2 Interpretable query plan selection and execution.* After extracting features, the feature vector $\mathbf{v}$ of a query plan $p$ is sent to EF to predict the execution time $t$. Suppose there are $l$ DTs $\{T_1, \ldots, T_l\}$ in EF. The decision paths of these $l$ DTs are output as a rule set $R = \{\phi_1, \phi_2, \ldots, \phi_n\}$. These interpretable rules provide full visibility into the decision-making process of EF. The final prediction result is obtained by taking the average of the predictions from the rules in $R$. Given a query $q_i$, the plan $p_i \in \mathcal{P}_i$ with the lowest predicted execution time $t_i$ is selected as the optimal plan $p_i^*$ and executed by the database engine.

**Example 2.** Consider Example 1. The rule set $R = \{\phi_1, \phi_2, \phi_3\}$ shown in Figure 1(a) is derived from the DTs in EF, where combination features in each rule, such as $Add(x_1, x_7)$ in $\phi_1$, are generated using the genetic programming algorithm. The predicted execution time for this query plan is calculated as the average of the predictions from decision rules $\phi_1, \phi_2, \phi_3$, which is $(5.63+1.02+2.35) \div 3 =$ 3s. This averaging mechanism is inherent to the evolutionary forest model, which combines multiple learned rules to produce a robust and stable prediction.

**Collect experiences for training.** After execution, each optimal plan $p_i^*$ and its actual execution time $t_i^{real}$ are recorded as an experience $exp_i : [p_i^*, t_i^{real}]$ and stored in the experience pool for training EF. To prevent unbounded growth of the experience pool and excessive memory consumption, the pool has a fixed capacity, denoted as $|E|$, allowing it to retain only the most recent $|E|$ experiences. When the number of experiences exceeds $|E|$, the oldest experiences are discarded. During training, we sample $m$ experiences $\{exp_1, \ldots, exp_m\}$ to generate training samples. Specifically, the plan $p_i^*$ from each $exp_i$ is fed into the feature extraction module to generate $\mathbf{v}_i$. The training dataset $\mathcal{D}_{\text{train}} = \{exp'_1, \ldots, exp'_m\}$ is then formed by collecting each pair $[\mathbf{v}_i, t_i^{real}]$ as $exp'_i$ from $m$ experiences, where $i \in \{1, \ldots, m\}$.

*3.3.3 Hint set update.* We utilize feature importance derived from the EF to update the hint sets for improved predictions. The feature importance derived from EF lists the combination features $Comb(x_i)$ that significantly impact the prediction results, along with their importance scores $\text{Imp}_{Comb(x_i)}$. To create a new hint set, we need to include both a scan and a join operator. We choose the operator from the parent-child nodes $(p_j, c_k)$ because it is the only type of node that contains both a scan and a join operator, enabling

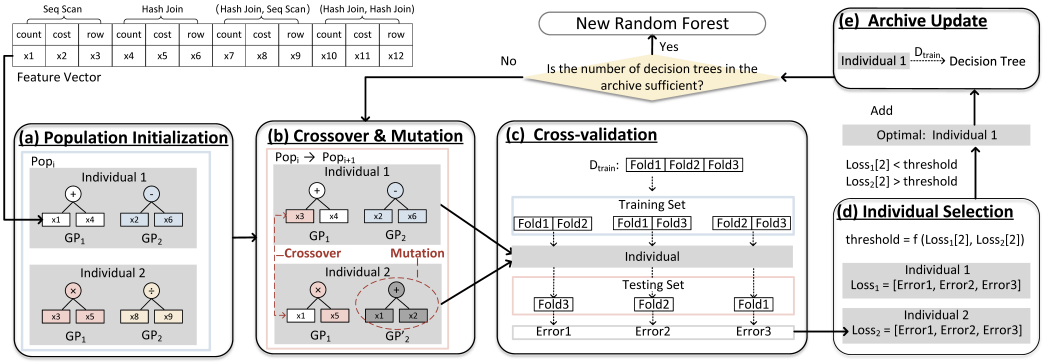



**Figure 3: Evolutionary forest generation.**

the formation of a valid hint set. Thus, we create new hint sets by evaluating the importance of $(p_j, c_k)$ according to $\text{Imp}_{Comb(x_i)}$.

**Node Type Importance Calculation.** To calculate the importance values of $(p_j, c_k)$, we first compute $\text{Imp}_{Comb(x_i)}$. These individual importance values are then grouped and summed to determine the overall importance of the corresponding operator. Next, we calculate the importance of each node type by assuming equal contribution from each $x_i$ in $Comb(x_i)$. The importance $\text{Imp}_{x_i}$ of each $x_i$ is computed by evenly distributing the score of $\text{Imp}_{Comb(x_i)}$: $\text{Imp}_{x_i} = \frac{\text{Imp}_{Comb(x_i)}}{|Comb(x_i)|}$, where $|Comb(x_i)|$ represents the number of $x_i$ in $\text{Imp}_{Comb(x_i)}$. Finally, we sum the importance of all $\text{Imp}_{x_i}$ associated with $(p_j, c_k)$ to calculate its importance.

**Augmented Hint Sets Generation.** The operators from parent-child node $(p_j, c_k)$ with the highest importance score are selected to construct two enhanced hint sets $H_{aug1}$ and $H_{aug2}$. Since these operators have a significant impact on the prediction results, they can be regarded as a potential set of hints to assist the query optimizer in generating better query plans. Specifically, $H_{aug1}$ includes only the operators in $(p_j, c_k)$, while $H_{aug2}$ represents their complementary set. Both sets are evaluated to explore potentially better query plans. As feature importance fails to capture the precise correlation between operator types and prediction, it remains unclear whether the impact of these operators is beneficial or detrimental. Given that operators in the $H_{aug1}$ may lead to suboptimal query plans, and operators beyond $H_{aug1}$ (referred to as $H_{aug2}$) might help generate better plans, it is essential to introduce the $H_{aug2}$ to supplement the candidate hints with additional promising operators. Note that adding two hint sets to the existing $\mathcal{H}$ does not significantly increase model overhead, but the potential performance gains from better query plans can be substantial.

**Example 3.** Consider Example 1. Figure 1(b) illustrates the feature importance associated with these four nodes, where $x7$, $x8$, and $x9$ are features of the node type (*Hash Join, Seq Scan*). The feature importance of (*Hash Join, Seq Scan*) is calculated as follows: $\text{Imp}_{(Hash\ Join, Seq\ Scan)} = \text{Imp}_{x_7} + \text{Imp}_{x_8} + \text{Imp}_{x_9} = \frac{0.08}{3} + \frac{0.22}{2} + \frac{0.15}{2} = 0.21$. Suppose its feature importance is the highest, *Hash Join* and *Seq Scan* is used to generate the enhanced hint sets $H_{aug1}$ = {*Hash Join, Seq Scan*}, $H_{aug2}$ = {*Bitmap Index Scan, Index Scan, Index Only Scan, Nest Loop, Merge Join*}. Then, $H_{aug1}$ and $H_{aug2}$ are added to the original $\mathcal{H}$ to help generate better query plans in the next stage.

## 3.4 Evolutionary Forest Generation

In this section, we describe the EF generation as a cyclic process consisting of five steps: population initialization, crossover and mutation, cross-validation, individual selection and archive update.

**Population Initialization.** The genetic programming algorithm starts by initializing a population that will serve as the foundation for subsequent evolution. This population, represented as $Pop = \{I_1, I_2, \ldots, I_n\}$, is established where each individual $I \in Pop$ represents the complete set of features required for constructing a DT. Specifically, $I = \{\text{GP}_1, \ldots, \text{GP}_y\}$, where each genetic program $\text{GP}_i \in I$ is a combined feature generated by selecting various $x_i$ from $\mathbf{v}$ and combining them with a function from set $F = \{+, -, \times, \div\}$ using the half-and-half method [7].

**Crossover and Mutation.** Crossover and mutation introduce genetic diversity and enable effective exploration of the solution space. The crossover operation involves swapping nodes between two GPs from different individuals, while the mutation operation replaces $\text{GP}_i$ with a newly generated $\text{GP}'_i$. After these updates are completed, the population transitions from $\text{Pop}_i$ to $\text{Pop}_{i+1}$.

**Cross-validation.** To evaluate the quality of an individual $I$, a fitness vector for $I$ is constructed via $K$-fold cross-validation towards $D_{train}$. In this method, $D_{train}$ is divided into $K$ folds, and then $K-1$ folds for training and one fold for test. For each $K-1$ training folds, a DT $T$ is trained given $I$ as its features and its performance is measured by the absolute deviation error $Error = |T(\mathbf{v}) - t^{real}|$, where $T(\mathbf{v})$ is the prediction towards $\mathbf{v}$ of the test fold. The mean of all prediction errors within the test fold is then computed to represent the error $Error$ of that specific fold. After completing $K$ iterations, the fitness vector for $I$ is formed as $Loss = [Error_1, \ldots, Error_K]$.

**Individual Selection.** To select the optimal individual $I_{\text{optimal}}$ from $Pop$, we employ the lexicase selection operator [9] combined with fitness vector. First of all, a threshold is calculated based on a selected element from the fitness vectors of all individuals in the same dimension. This threshold is then used to filter out individuals whose values in that dimension exceed the threshold. The process is

repeated across the remaining dimensions to progressively narrow down the candidate set until $I_{\text{optimal}}$ is identified.

**Archive Update.** In this step, a DT is trained with $I_{\text{optimal}}$ using $\mathcal{D}_{\text{train}}$ and then stored in an archive. This archive holds a collection of DTs that will be used to construct the final EF. The archive's capacity is set to match the predefined number of trees in EF. Once the archive reaches its full capacity, a new EF is constructed by aggregating the stored DTs, which replaces the previous EF, therefore concluding the evolutionary process. If the archive is not yet full, the algorithm continues to the crossover and mutation step to evolve *Pop* into the next generation.

**Example 4.** As illustrated in Figure 3, consider $Pop_i = \{I_1, I_2\}$, where both $I_1$ and $I_2$ contains $\{\text{GP}_1, \text{GP}_2\}$, In $I_1$, $\text{GP}_1 = x_1 + x_4$ and $\text{GP}_2 = x_2 - x_6$. In $I_2$, $\text{GP}_1 = x_3 \times x_5$ and $\text{GP}_2 = x_8 \div x_9$. Each GP is a symbolic expression that maps input features (e.g., operator counts, costs) to a predicted execution time or cost adjustment. The goal is to evolve GPs that produce accurate and interpretable predictions. During the crossover operation, a subtree (here, a single node) is exchanged between individuals to promote genetic diversity. Specifically, node $x_3$ from $\text{GP}_1$ in $I_1$ is swapped with node $x_1$ from $\text{GP}_1$ in $I_2$. This results in updates where $\text{GP}_1$ in $I_1$ becomes $\text{GP}_1 = x_3 + x_4$, and in $I_2$ it updates to $\text{GP}_1 = x_1 \times x_5$. This operation mimics biological recombination and helps explore new rule combinations without starting from scratch. To further increase diversity and avoid premature convergence, a mutation is applied to $\text{GP}_2$ of $I_2$, replacing $\text{GP}_2 = x_8 \div x_9$ with a new generated $\text{GP}'_2 = x_1 + x_2$. Mutation introduces novel structures into the population, enabling the discovery of potentially better-performing rules. Next, for $I_1$, three-fold cross-validation is performed by partitioning $D_{train}$ into three folds: $\{Fold1, Fold2, Fold3\}$. In each round, two folds are used for training while one fold is reserved for validation. This process cycles through all subsets to obtain the loss values represented as $Loss_1 = [Error1, Error2, Error3]$. Similarly, the loss $Loss_2$ for $I_2$ is obtained. Based on a threshold computed from the second dimension of all fitness vectors, only $Loss_1[2]$ of $I_1$ meets the criterion. Therefore, $I_1$ is selected as the optimal individual, a DT generated according to $I_1$ is stored in the archive. Over successive generations, multiple such DTs are evolved and combined into the final evolutionary forest, forming an ensemble that balances accuracy, interpretability, and generalization.

## 3.5 Evolutionary Forest Pruning

To reduce inference latency and improve interpretability, pruning strategies are employed to decrease the number and depth of DTs during EF generation, resulting in a simplified and pruned model.

*3.5.1 Reduce Number of DTs.* Each DT contributes a specific rule for a prediction. By reducing the number of DTs via clustering, we can decrease the total number of rules, which enhances the interpretability of the model. Additionally, having fewer DTs allows for faster model inference. Existing studies often predefine the number of clusters before performing clustering [33]. However, due to the complexity and diversity of real-world datasets, it is challenging to accurately determine an appropriate cluster size in advance. An unsuitable cluster size may either discard high-performing DTs or introduce redundant ones, ultimately affecting the prediction accuracy. Thus, we employ an Unsupervised Optimal Path Forest clustering algorithm (UOPF) [18, 19], which automatically learns the optimal number of clusters based on the dataset characteristics.

**Decision tree clustering.** In this process, we group all the DTs in EF into distinct clusters based on the errors obtained during the cross-validation phase. Specifically, for each $I$, we also record the absolute deviation error *Error* for each exp′ in every fold. The errors from all $\text{exp}' \in \mathcal{D}_{train}$ are then combined into a vector $\mathbf{e}$ for $I$. The DTs generated in EF, along with the corresponding vector $\mathbf{e}$ associated with each DT's individual, are then used as inputs for the UOPF algorithm. This algorithm subsequently clusters the DTs into different groups.

**Select the Representative DT from Each Cluster.** After clustering, the DTs is divided into groups. For each group, the error is computed as the mean error of each DT is this group. The average value of $\mathbf{e}_i$ is computed by summing all its elements and dividing by the total number of elements. The tree with the lowest value is selected as the representative. A new DT set is then constructed by aggregating these representative trees from all clusters.

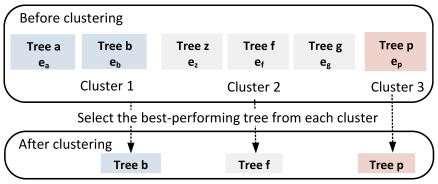



**Figure 4: Reducing the number of DTs via UOPF clustering.**

**Example 5.** As illustrated in Figure 4, the original EF consist of six DTs {a, b, z, f, g, p}. After clustering, these DTs are grouped into three categories {a, b}, {z, f, g}, and {p} based on the similarity of their decision paths and prediction behaviors. From each group, a representative DT is selected, resulting in the new set of representative trees {b, f, p}. This new ensemble forms the pruned EF, which retains the predictive power of the original forest while significantly reducing model complexity.

*3.5.2 Reduce the Depth of DTs.* In each DT, nodes are arranged at different depths based on their feature importance, with deeper nodes contributing less to prediction accuracy. While increasing the depth of a DT lends to slight improvements in accuracy, it also results in exponential increase in the number of nodes and a significant reduction in model interpretability. Thus, we introduce an optimization framework [12] to reduce the depths of DTs.

**Least Squares Objective Function.** Pruning deep nodes in decision trees can be modeled as a regularized least squares optimization problem, as described in Eq. (1).

$$\min_{r_1, r_2, \ldots, r_l} \quad \frac{1}{m} \left\| \mathbf{y} - \frac{1}{l} \sum_{i=1}^{l} \mathbf{D_i r_i} \right\|_2^2 + \alpha \sum_{i=1}^{l} \frac{\|\mathbf{A_i r_i}\|_1}{d_i}, \quad \forall i \in [l]. \tag{1}$$

Here, the term $\frac{1}{m} \left\| \mathbf{y} - \frac{1}{l} \sum_{i=1}^{l} \mathbf{D_i r_i} \right\|_2^2$ represents a least squares objective that measures the reconstruction error between the ensemble prediction and the ground truth. $\mathbf{y}$ is a vector consisting of the actual execution times of the samples. The variable $\mathbf{r_i}$ represents how many layers to retain in the $i$-th DT and serves as the optimization

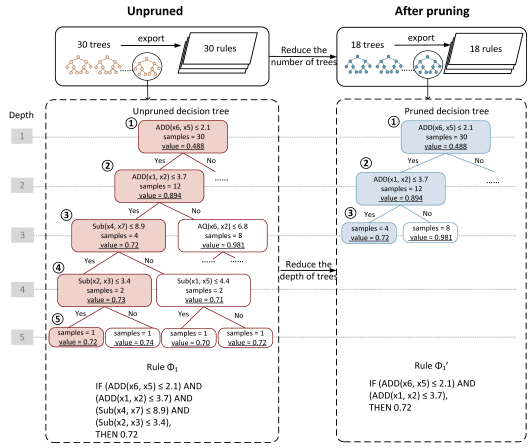



**Figure 5: An example of output rules after reducing the number and depth of DTs in EF.**

target in the formulation. $\mathbf{D_i}$ is constructed with each row representing a training sample, capturing the prediction increments across depth layers through differences in predicted means. The regularization term $\alpha \sum_{i=1}^{l} \frac{\|\mathbf{A_i r_i}\|_1}{d_i}$ helps reduce overfitting and encourages simpler trees by pruning less important layers. The parameter $\alpha$ controls the strength of regularization, where larger values lead to shallower trees. The weight matrices $\mathbf{A_i}$ helps determine which layers are more important so that the pruning process can produce ensembles that are both compact and efficient. $d_i$ represents the depths of the unpruned decision trees.

**Optimization Algorithm.** Inspired by the cyclic coordinate descent method [23], we utilize a randomized block-wise update strategy to solve the optimization problem. It iteratively refines variable blocks $\mathbf{r_i}$ based on residual errors, which enhances convergence stability and helps avoid suboptimal solutions. Unlike traditional cyclic coordinate descent, which follows a fixed sequential order for block updates, our approach uses random selection to improve robustness and reduce the risk of getting trapped in local optima due to deterministic update patterns. This bottom-up hierarchical solution converges quickly, thus it does not significantly increase the training time of EF. It effectively decreases the depths of the DTs in the EF with only a slight reduction in performance, resulting in interpretable and lightweight DTs that enhance transparency.

**Example 6.** As shown in Figure 5, the generated EF initially contains 30 trees. After pruning, the number of DTs decreases to 18. For a specific DT in the original EF, the rule $\Phi_1$ is obtained from the path leadning from Node 1 to Node 5, resulting in a prediction of 0.72. It is observed that the nodes at the bottom of the DT typically contain only one or two samples and have low feature importance. Consequently, during pruning, nodes at depths 4 and 5 are removed. The updated rule $\Phi_1'$ is derived from the path from Node 1 to Node 3. The value at Node 3 also provides the prediction of 0.72, ensuring that the prediction remains accurate. The updated rule excludes the items associated with Node 3 and Node 4, thereby enhancing the interpretability of the decision rules.

# 4 EXPERIMENTS

In this section, we evaluate the performance of LIO across various dimensions. First, we compare the total runtime, tail latency, and generalization ability of LIO against baseline approaches (Sections 4.2-4.4). Next, we evaluate the influence of pruning on LIO's prediction accuracy and interpretability (Sections 4.5). Third, we analyze the impact of hint set selection and the impact of the encoded count feature (Sections 4.6-4.7). Finally, we study the performance of LIO on complex queries (Section 4.8).

## 4.1 Experimental setup

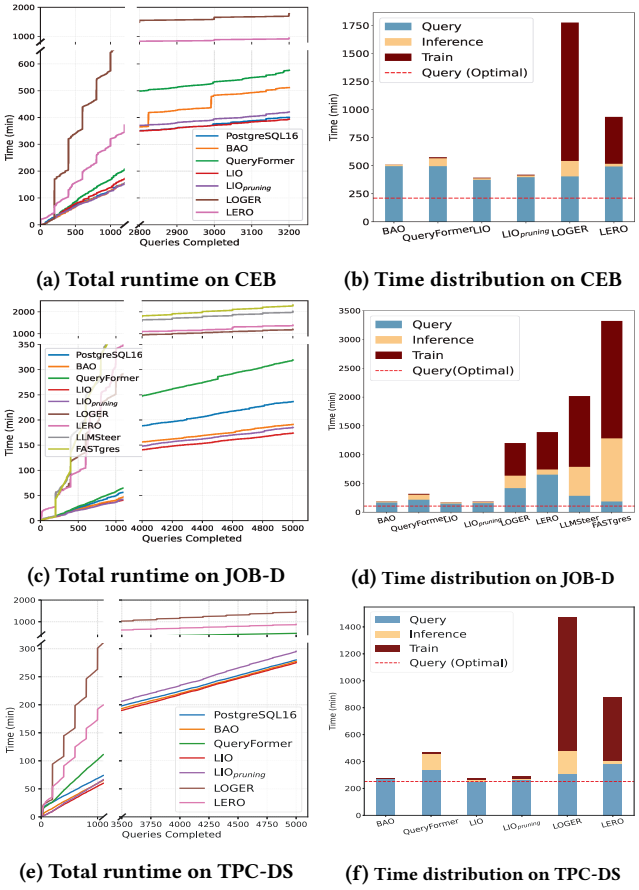
**Environment.** The CPU used is an Intel(R) Xeon(R) CPU E5-2680 v4 @ 2.40GHz, featuring 28 physical cores and 56 threads. The system is equipped with 64GB of RAM and four NVIDIA GeForce RTX 3090 GPUs. The operating system is Ubuntu 24.04 LTS. We implement LIO using Python 3.8 and PyTorch 2.4.1. To enable or disable hints for PostgreSQL's query optimizers, we execute SET commands as part of the LIO implementation.

**Datasets and Workloads.** We conduct experiments using seven workloads across two real-world datasets: IMDB, TPC-DS [20]. For IMDB, we use four workloads: *Join Order Benchmark (JOB)*, *JOB-Extend*, *CEB*, and *JOB-D*. The JOB workload [11] consists 113 queries and is commonly used in most experiments. JOB-Extend is a workload created by NEO [14], which includes 24 queries that have completely different join relationships and predicates compared to JOB. The CEB workload [17], derived from JOB queries with long execution times, contains 3,133 queries generated based on 16 templates. Additionally, JOB-D is a dynamic workload generated by HybridQO [29], from which we randomly select 5,000 queries for evaluation. This workload simulates real-world execution environments and is characterized by frequent and complex variations in queries. For *TPC-DS*, a standard benchmark with data and query generator, we set the scale factor to 22 to generate the data. The standard workload consists of 99 query templates. It is important to note that most TPC-DS query templates are not SPJ (Select-Project-Join) templates, which are generally unsupported by most baseline methods. Therefore, to facilitate comparison with these baseline methods, we followed the LOGER configuration and excluded overly simplistic templates and those not compatible with the pg_hint_plan hook function, resulting in the retention of 20 templates. These templates are numbered 3, 7, 12, 18, 20, 26, 27, 37, 42, 43, 50, 52, 55, 62, 82, 84, 91, 96, 98, and 99. Each template generates 250 queries, leading to a total of 5,000 queries. Furthermore, to evaluate the performance of LIO on arithmetic and aggregated queries, we utilize the DSB benchmark [3], which is built upon the TPC-DS benchmark to generate workloads. Queries are generated from all the templates and classified by SQL keywords. Among these, 764 arithmetic queries and 829 aggregated queries are selected as workloads *DSB-AQ* and *DSB-AGQ*, respectively.

**Compared methods.** We compare our proposed method LIO with seven state-of-the-art baselines. (1) PostgreSQL 16.9. We use the following configuration parameters fixed across all workloads: the parameter work_mem sets to 4 MB, shared_buffers to 128 MB, effective_cache_size to 4 GB, and geqo_threshold to 12. (2) BAO [13]. BAO uses hint sets $\mathcal{H}$ to guide the existing database optimizer

in generating multiple query plans and employs a TCNN to predict their execution times. In our experiments, we set the number of hint sets to 5. (3) QueryFormer [30]. QueryFormer develops a tree-structured Transformer model to learn the representation of physical query plans for a variety of machine learning tasks. (4) Lero [32]. Based on plan exploration and pairwise plan comparison, Lero learns the difference between plan pairs and learns to improve the end-to-end quality of query optimization. (5) LOGER [2]. LOGER is a learned optimizer that employs a graph Transformer to capture information from tables and predicates within the join graph. (6) Fastgres [26]. FASTgres proposes a context-aware classification strategy for predicting hint sets directly for incoming queries. (7) LLMSteer [1]. LLMSteer utilizes the LLM *text-embedding-3-large* to embed raw SQL and trains a supervised learning model on a small labeled dataset to predict the optimal query hint. For the proposed method, we use LIO and LIO$_{pruning}$ for comparsion. LIO uses the evolutionary algorithm to generate and update RF and does not use the pruning strategies. LIO$_{pruning}$ ncorporates pruning strategies during the generation of EF. An acceptable error growth threshold is set to $\eta = 0.05$ unless otherwise specified by adjusting the hyperparameter $\alpha$ in Eq. (1) within the training set. The objective is to prune as many layers and nodes as possible. The validation set is used to evaluate whether the error growth remains within the specified $\eta$ range, which helps determine the final model.

**System Hyperparameters.** In the experiments, the system hyperparameters for all methods are set as $N$=200, $f$=200, and $|E|$=5000, where $N$ denotes the number of experiences used for training, and $f$ indicates the number of queries processed before retraining the model. $|E|$ specifies the maximum number of experiences retained, and older experiences are discarded following a first in first out policy to control storage usage. After every 200 queries, 200 experiences are randomly selected from the experience pool as the dataset to train or iterate the model. The experience pool retains only the latest 5000 experiences. During cross-validation of EF generation, $K$ is set to 5. $\mathcal{H}$ is initialized with 5 carefully selected $H_i$.




(a) Total runtime on CEB (b) Time distribution on CEB

(c) Total runtime on JOB-D (d) Time distribution on JOB-D

(e) Total runtime on TPC-DS (f) Time distribution on TPC-DS

**Figure 6: Performance comparison across different datasets: the left column shows the evolving total runtime, while the right column details the contributions of training, inference, and query execution time to the overall runtime. Query (Optimal) represents the optimal total execution time, achieved by always selecting the best hint set for each query.**

## 4.2 Total Runtime of Workloads

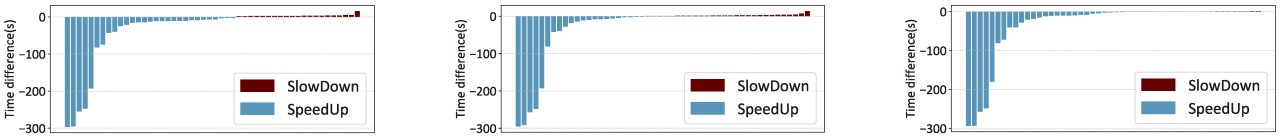
Figure 6 presents the overall performance comparison across different datasets. In terms of total runtime, the proposed method LIO achieves the fastest query processing on all datasets, with consistent and meaningful improvements. On the CEB dataset, LIO reduces the total runtime by 2% compared to the best baseline PostgreSQL 16, and by 23% compared to the strongest learning-based baseline Bao. On the Job-D dataset, where Bao is the top-performing baseline, LIO further reduces its total runtime by 9%. On the TPC-DS dataset, Bao again achieves the best performance among baselines, and LIO improves upon it by reducing the total runtime by 1%. Among the baseline methods, PostgreSQL 16 surprisingly outperforms most learning-based approaches except Bao. This suggests that, as PostgreSQL's built-in query optimizer continues to advance, many existing learning-based optimizers are no longer able to consistently surpass its default performance. In contrast, Bao is built on top of PostgreSQL and leverages plan-level feedback, which helps it maintain strong competitiveness.

In terms of time distribution, LIO performs best due to the use of EF, which strike a good balance among training time, inference time, and prediction accuracy. LIO achieves the shortest query execution time among all methods. On the TPC-DS dataset, LIO's query execution time nearly reaches the level of Query (Optimal), demonstrating its ability to accurately predict high-quality query plans. Notably, even though LIO$_{pruning}$ reduces the number and depth of decision trees in the EF, thereby improving training efficiency, it still achieves shorter query execution time than all baseline methods.

Among the baselines, Bao benefits from a short inference time due to its use of tree-structured feature representations and efficient training with TCNN, making it simple and fast. However, it suffers from lower prediction accuracy compared to LIO. In contrast, QueryFormer and LOGER employ complex Transformer-based models, resulting in high inference latency, which negatively impacts their overall performance. Regarding training time, LOGER, Lero, LLMSteer, and FASTgres require execution feedback for all possible hint sets to construct training labels, leading to prohibitively long label generation and training overhead. Moreover, the inference phases of LLMSteer and FASTgres are also very slow. LLMSteer

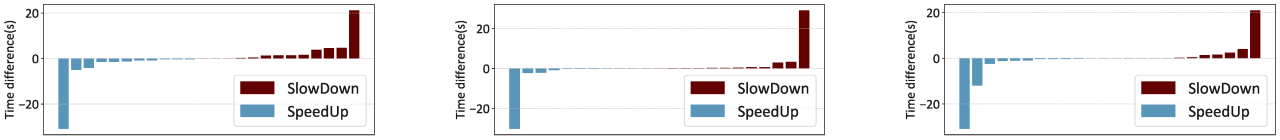
**Table 1: Percentile latency (seconds) for each method on three datasets.**

| Method | CEB | | | | | JOB-D | | | | | TPC-DS | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 30th | 50th | 90th | 95th | 99.5th | 30th | 50th | 90th | 95th | 99.5th | 30th | 50th | 90th | 95th | 99.5th |
| BAO | 0.594 | 0.603 | **0.975** | 1.163 | 1.356 | 0.390 | 0.686 | 1.837 | 1.997 | 0.974 | 0.920 | 0.977 | **1.013** | 0.997 | 0.994 |
| QueryFormer | 0.406 | 0.515 | 0.659 | 0.795 | 0.916 | 0.142 | 0.257 | 1.159 | 1.046 | **1.392** | 0.329 | 0.582 | 0.647 | 0.754 | 0.780 |
| LIO | 0.706 | 0.659 | 0.948 | **1.231** | **1.431** | 0.406 | 0.726 | **1.912** | **2.183** | 1.276 | **0.967** | **0.997** | 0.994 | **1.028** | **1.026** |
| $LIO_{\eta=0.05}$ | **0.744** | **0.702** | 0.943 | 1.099 | 1.198 | **0.423** | **0.740** | 1.872 | 2.113 | 1.304 | 0.599 | 0.979 | 0.964 | 1.018 | **1.026** |
| LOGER | 0.279 | 0.433 | 0.803 | 0.936 | 0.966 | 0.073 | 0.155 | 0.710 | 0.823 | 0.101 | 0.318 | 0.628 | 0.723 | 0.754 | 0.726 |
| LERO | 0.308 | 0.433 | 0.915 | 1.102 | 0.931 | 0.216 | 0.215 | 0.705 | 0.771 | 0.101 | 0.329 | 0.714 | 0.765 | 0.852 | 0.850 |
| LLMSteer | – | – | – | – | – | 0.034 | 0.078 | 0.766 | 0.929 | 0.280 | – | – | – | – | – |
| FASTgres | – | – | – | – | – | 0.015 | 0.035 | 0.438 | 0.686 | 0.687 | – | – | – | – | – |
| PostgreSQL 16 | 1.056s | 2.169s | 13.114s | 26.774s | 190.198s | 0.218s | 0.535s | 8.202s | 13.734s | 30.408s | 1.175s | 3.544s | 6.245s | 8.143s | 11.172s |




**(a) BAO** **(b) QueryFormer** **(c) LIO**

**Figure 7: The difference in query execution time of PostgreSQL on JOB.**




**(a) BAO** **(b) QueryFormer** **(c) LIO**

**Figure 8: The difference in query execution time of PostgreSQL on JOB-Extend.**

requires extensive feature preprocessing before feeding inputs into the LLM. FASTgres involves query classification, context parsing, feature extraction, encoding, and model prediction, and each step adds significant latency. As a result, both methods exhibit long total runtimes. On the CEB and TPC-DS datasets, they fail to complete within 50 hours and are therefore not included in the figures.

## 4.3 Tail Latency Analysis

Table 1 presents the percentile latencies of all the methods, expressed as ratios relative to the corresponding percentile latencies of PostgreSQL. A ratio less than 1 indicates slower execution speed than PostgreSQL 16. The last row shows the actual latency of PostgreSQL 16. Overall, regression occurred in queries with lower percentile latencies (e.g., 30th). This indicates that for simpler queries, learning-based optimization methods introduce additional inference overhead, increasing overall query latency and diminishing their advantage. This issue is not unique but represents an inherent bottleneck faced by many learning-based query optimizers.

For queries with higher percentile latencies, the latency of LIO and $LIO_{pruning}$ is lower than that of PostgreSQL on all datasets. This is especially true for complex queries with longer execution times, where methods achieve greater improvements. As query complexity increases, the performance gains of the methods become more substantial, notably improving query efficiency. These latency reductions in complex queries offset the earlier mentioned regression and enable LIO and $LIO_{pruning}$ to achieve faster execution times than PostgreSQL 16. For TPC-DS, it can be seen that $LIO_{pruning}$ achieves good results, performing similarly to LIO. This suggests that pruning may not always affect the optimization ability across different types of queries and scenarios. In most cases, LIO outperforms the baselines on all datasets. Since CEB is more complex and has longer execution times, this demonstrates that LIO is good at handling and optimizing tail-end queries.

## 4.4 Absolute Generalization Performance

This experiment evaluates the absolute performance of Bao, QueryFormer and LIO. The models are trained on JOB-D and then tested on both JOB and JOB-Extend. A positive value indicates slower execution compared to PostgreSQL, reflecting performance regression. JOB serves as the template for generating JOB-D, and the queries in both datasets share similar join structures. This similarity makes them well-suited for assessing absolute performance under closely matched workload conditions. As shown in Figure 7, LIO reduces execution time for a larger number of queries than Bao and QueryFormer on the JOB dataset, while incurring fewer instances of performance regression. These results demonstrate

**Table 2: Comparison of accuracy and interpretability of LIO with pruned and unpruned.**

| Method | CEB | | | | JOB-D | | | | TPC-DS | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Median $R^2$ | Mean $R^2$ | Number | Depth | Median $R^2$ | Mean $R^2$ | Number | Depth | Median $R^2$ | Mean $R^2$ | Number | Depth |
| LIO | **0.466** | 0.464 | 100 | 7.22 | **0.896** | **0.887** | 100 | 6.90 | **0.974** | **0.971** | 100 | 7.46 |
| $\text{LIO}_{\eta=0.01}$ | 0.458 | 0.453 | 87 | 5.12 | 0.887 | 0.878 | 78 | 5.68 | 0.972 | 0.967 | 68 | 5.23 |
| $\text{LIO}_{\eta=0.025}$ | 0.446 | **0.477** | 72 | 4.39 | 0.838 | 0.830 | 54 | 4.48 | 0.970 | 0.963 | 51 | 4.11 |
| $\text{LIO}_{\eta=0.05}$ | 0.440 | 0.476 | **59** | **3.25** | 0.815 | 0.814 | **42** | **3.98** | 0.967 | 0.964 | **48** | **3.68** |

the effectiveness of the genetic programming algorithm in improving model accuracy and underscore its critical role in guiding the evolution of high-quality decision models.

The queries in JOB-Extend feature entirely different join structures and represent previously unseen workloads. They are designed to evaluate the generalization capability of models when faced with tasks involving unfamiliar data distributions. As shown in Figure 8, all three methods exhibit some degree of performance regression, underscoring the significant room for improvement in cross-workload generalization. Among them, LIO shows minimal regression and reduces execution time for a larger number of queries, demonstrating its superior ability to select high-quality query plans under distribution shift.

## 4.5 Model Accuracy and Interpretability

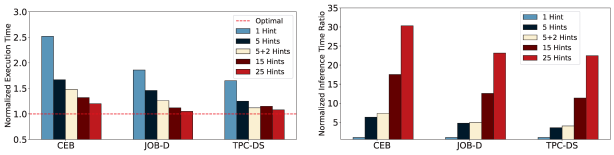
In this experiment, datasets were used for multiple rounds of training, validation, and testing. $R^2$ was recorded to measure the model's data-fitting ability. Additionally, the number and the depth of DTs were compared as interpretability metrics. The experimental results are shown in Table 2.

**Dataset Analysis.** We first analyze the experimental results across the three datasets. The CEB dataset exhibits the lowest $R^2$ values, indicating that it has a more complex data distribution, making it difficult for models to learn and fit the distribution effectively. After the pruning operation, interpretability is at its lowest, meaning that these models tend to have the largest tree count and depth overall. In contrast, for the TPC-DS dataset, the $R^2$ values among different models show little variance and are very close to 1, indicating superior model fitting capability. This further suggests that the TPC-DS dataset has a simpler structure, making it easier to model. Under the same training setup, the data distribution of a dataset significantly influences the model's fitting ability and interpretability.

**Comparison of Model Performance.** From the perspective of model fitting ability, the tables show that LIO achieves the highest median $R^2$, indicating that LIO has the best fitting capability. Further analysis of the box plots reveals that LIO also has the smallest interquartile range for $R^2$, demonstrating the lowest performance fluctuation across multiple experimental runs and the highest stability. As the error threshold $\eta$ increases, the $R^2$ values of the $\text{LIO}_{\eta=0.01}$ models gradually decrease, and the interquartile range increases, suggesting that pruning reduces the model's ability to fit the data, thereby lowering its explanatory power and impacting both predictive accuracy and generalization performance. From the perspective of interpretability, as the error threshold $\eta$ increases, models contain fewer DTs and shallower depths, resulting in a smaller ensemble size. The $\text{LIO}_{\eta=0.05}$ model exhibits the highest interpretability among all models. However, its $R^2$ is relatively low, indicating that improving interpretability requires reducing the number and depth of DTs, which may weaken model fitting ability. Compared to LIO, the $R^2$ value of $\text{LIO}_{\eta=0.01}$ does not show a significant drop, with a difference of less than 0.1, and the interquartile range of $R^2$ is also nearly identical, reflecting that $\text{LIO}_{\eta=0.01}$ does not suffer a noticeable loss in data-fitting ability. At the same time, the average tree depth in $\text{LIO}_{\eta=0.01}$ is reduced by 25%, and the total number of DTs is reduced by 20%, making the model structure more compact and significantly enhancing interpretability. This demonstrates that, under the current pruning strategy, it is possible to achieve a good balance between interpretability and fitting ability, allowing the model to maintain strong predictive and generalization capabilities while improving interpretability.

## 4.6 The Impact of Initial Hint Selection

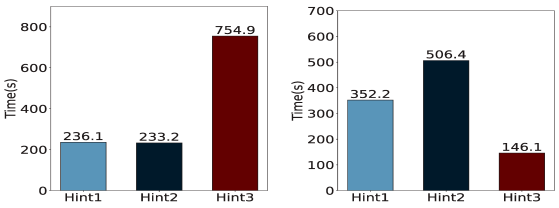
In this experiment, we explore the impact of inital hint selection by varying different number of hint sets and different operator combinations in the hint sets.




**(a) Query Execution Time** **(b) Model Inference Time**

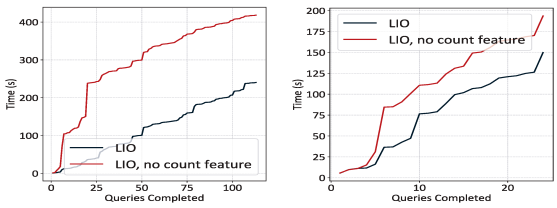
**Figure 9: The impact of different number of $H_i$.**

**The impact of different hint set number.** We first conduct experiments on the CEB, JOB-D, and TPC-DS with varying $H_i$ numbers collected execution and inference times for 200 queries. As shown in Figure 9(a), "Optimal" represents the ideal execution time, normalized to 1. Increasing $H_i$ improves query plan quality, but the benefit is saturated. However, the optimization effect saturates as $H_i$ continues to increase, suggesting that selecting representative $H_i$ can significantly reduce execution time. LIO's two enhanced $H_i$ improve query plans beyond five $H_i$, achieving performance close to 15 $H_i$ on TPC-DS. As shown in Figure 9(b), the inference time of a model with one $H_i$ is used as the baseline, with other values presented as relative ratios. The baseline time is normalized to 1, providing a clear comparison of the impact of different $H_i$ numbers on inference time. As $H_i$ increases, inference time grows almost proportionally. LIO selects five $H_i$ along with two enhanced $H_{aug}$, leading to a slightly higher inference time than using five $H_i$ alone, but without significant overhead. This configuration improves query plan quality, effectively reducing execution time.



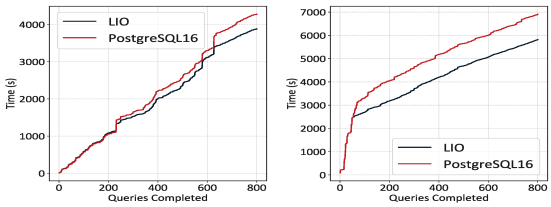

(a) JOB (b) JOB-Extend

**Figure 10: The impact of different initial $H_i$.**




(a) JOB (b) JOB-Extend

**Figure 11: The impact of the encoded counts feature.**




(a) DSB-AQ (b) DSB-AGQ

**Figure 12: Total runtime on DSB.**

**The impact of initial hint selection.** The five hint sets used in our experiments are: Case 0: {'hashjoin', 'indexscan', 'mergejoin', 'nestloop', 'seqscan', 'indexonlyscan'}; Case 1: {'hashjoin', 'indexonlyscan', 'indexscan', 'mergejoin', 'seqscan'}; Case 2: {'hashjoin', 'indexonlyscan', 'nestloop', 'seqscan'}; Case 3: {'hashjoin', 'indexonlyscan', 'seqscan'}; Case 4: {'hashjoin', 'indexonlyscan', 'indexscan', 'nestloop', 'seqscan'}. We select three representative combinations to study the impact of the initial hint selection: Hint 1 = {Case 0, Case 1, Case 3}, Hint 2 = {Case 0, Case 2, Case 3}, and Hint 3 = {Case 0, Case 2, Case 4}. Figure 10 shows the execution time of LIO on the JOB and JOB-Extend datasets. On the JOB dataset, Hint 3 results in the longest execution time, suggesting that reintroducing indexscan and nestloop in this configuration does not improve performance and may even lead to suboptimal plan selections under familiar workloads. In contrast, on JOB-Extend, which contains queries with unseen and more complex join patterns, Hint 3 achieves the shortest execution time, indicating that the inclusion of indexscan and nestloop plays a critical role in enabling effective plan exploration for complex, out-of-distribution queries. These results highlight the importance of carefully designing the initial hint space. While certain operators may be detrimental in simple or familiar scenarios, they can become essential for generalization to more complex and previously unseen workloads, and thus we include all five hint sets as our intial hint sets.

## 4.7 The Impact of the Encoded Count Feature

In this part, we evaluate the impact of the encoded count feature in LIO on the JOB and JOB-Extend datasets. Results of query execution time are presented in Figure 11. On both datasets, we observe that when the count information is not encoded in the feature vector, the query execution time increases significantly. This indicates that operator counts are critical for effective plan selection. Without this information, the model struggles to distinguish between semantically similar but performance divergent plans, leading to suboptimal decisions and degraded runtime performance. The performance gap is particularly evident on the JOB dataset, where queries follow familiar patterns seen during training. This suggests that even for well-distributed workloads, the absence of operator counts impairs the model's ability to make reliable predictions.

## 4.8 The Performance on Complex Queries

In this experiment, we evaluate the performance of LIO on arithmetic and aggregation queries. Although LIO does not explicitly encode arithmetic or aggregation operators in its feature vector, it can still model the structural and cost-related characteristics of the query plan, enabling it to generalize to more complex query types. Since most learning-based query optimizers are designed and trained primarily on SPJ queries, they typically exclude complex arithmetic expressions and aggregate functions. Therefore, we use PostgreSQL 16 as the sole baseline for comparison. The results are presented in Figure 12. We can see that LIO outperforms PostgreSQL 16 on both arithmetic and aggregation queries in terms of total runtime. The performance gain is more pronounced on aggregation queries, indicating that LIO effectively leverages learned plan patterns to guide optimization, even without explicit semantic features for aggregation operations.

# 5 RELATED WORK

**Learning-based optimizers.** We compare learned query optimizers based on several key aspects, as summarized in Table 3.

Query plan representation enables learning the mapping from physical plans to execution times for efficient plan prediction. Methods like Neo [14], Bao [13], Balsa [28], and Lero [32] use Tree-CNNs to capture parent-child dependencies, while Loger [2] and Queryformer [30] employ Transformers to model long-distance dependencies via self-attention. These approaches often integrate database statistics (e.g., cardinality, cost, histograms) but yield complex representations that are computationally expensive to train. In contrast, Fastgres [26], LLMSTEER [1], and LLM-QO [24] bypass explicit plan encoding. Fastgres predicts hint sets via context-aware classification, while the latter two use large language models (LLMs) to generate plans directly. Unlike these, LIO encodes query plans into feature vectors and optimizes them using evolutionary forests. It captures parent-child dependencies by jointly modeling single-type and parent-child-type nodes, and includes operator count as a proxy for query complexity. By combining ensemble learning with

**Table 3: Comparison of existing learning-based optimizers.**

| Optimizer | Query Plan Representation | Plan Structure Encoding | Database Statistics | Interpretability | Training time | Integration with a real DBMS |
|---|---|---|---|---|---|---|
| Neo [14] | Tree-CNN | Parent-Children Dependency | Estimated card | No | Large | No |
| Bao [13] | Tree-CNN | Parent-Children Dependency | Estimated card, cost | Yes | Short | Yes |
| Balsa [28] | Tree-CNN | Parent-Children Dependency | Estimated card | No | Large | No |
| Lero [32] | Tree-CNN | Parent-Children Dependency | Estimated card | No | Large | Yes |
| LOGER [2] | Transformer | Parent-Children/Long-Distance Dependency | Estimated card | No | Large | No |
| Queryformer [30] | Transformer | Parent-Children/Long-Distance Dependency | Sample, histogram | No | Short | No |
| Fastgres [26] | NA | NA | NA | No | Large | No |
| LLMSTEER [1] | NA | NA | NA | No | Large | No |
| LLM-QO [24] | NA | NA | NA | No | Large | No |
| LIO (Ours) | Feature vector | Parent-Children/Long-Distance Dependency | Estimated card, cost, count | Yes | Short | Yes |

evolutionary search, LIO captures relational plan characteristics with high adaptability, low training cost, and strong interpretability.

Interpretability is important for fostering user trust and validating the optimizer's decision-making process. Among existing optimizers, Bao [13] offers a degree of interpretability: its EXPLAIN function recommends physical operators, which users can inspect to diagnose performance issues. However, it reveals neither the model's decision logic nor why specific operators were chosen, offering little actionable insight. In contrast, LIO explicitly quantifies operator importance and clarifies the reasoning behind its predictions. By using these interpretable results to guide hint selection, LIO enhances the transparency and decision-making clarity of ML-based query optimization.

Training time varies significantly across different approaches. LOGER, Fastgres, LLMSTEER, and LLM-QO operate under a supervised learning framework, requiring the collection of labels by retrieving optimal runtimes and corresponding hint sets for all queries in the training data. In contrast, LIO shares a similar training setup with Bao and Queryformer, as it only needs to execute the most likely query plan predicted by the model. This results in shorter training times for Bao, Queryformer and LIO.

Finally, integration with a real DBMS is crucial for the practical application of learning-based optimizers, as it affects their adoption in existing environments. BAO [13] and LIO enhance the native query optimizer by injecting hints to generate candidate execution plans. Lero [32] manipulates cardinalities to produce alternative plans and utilizes a pairwise classification model to select the most cost-effective option. Compared to BAO and Lero, LIO better leverages the expertise of seasoned optimizers during training, mitigating the risk of poor prediction accuracy in early stages that can lead to suboptimal query plans and significant performance degradation.

**Machine learning techniques for query-related tasks.** Machine learning (ML) techniques have enhanced various query optimization tasks, including cardinality and cost estimation [16, 22, 27], join order selection [15] and index selection [4]. Among these methods, LIO shares a similar approach with AIMeetsAI [4] by defining feature indicators for each physical operator and weighting them based on node height in the query plan tree. AIMeetsAI encodes query plans by collecting feature values and performing dimension-wise summation to create fixed-length vectors. Building on this, LIO enhances the feature space by introducing global indicators, such as the number of operators, to improve prediction accuracy. For interpretability, BayesCard [27] uses Bayesian networks with DAGs to model attribute dependencies, allowing users to verify structures and parameters against prior knowledge. In contrast, LIO ensures transparency through genetic programming. Its feature generation is traceable, and feature importance is quantifiable, enabling users to clearly identify the most influential predictors.

**Optimizer assistants.** Automated database system configuration tuning can also improve the workload performance by optimizing DBMS parameters [6, 8, 10, 25]. Recent research [6, 10] has increasingly focused on leveraging LLMs to extract tuning hints from documentation and other resources. For example, $\lambda$-Tune [6] uses LLMs to translate natural language guidance into configuration changes. While effective at hint extraction, these methods struggle with the combinatorial space of possible configurations, requiring extensive trials that limit efficiency. In contrast, LIO directly optimizes query execution plans, addressing a distinct and complementary aspect of database performance.

## 6 CONCLUSION

In this paper, we propose LIO, an interpretable learning-based query optimization framework that uses genetic programming to evolve random forests, effectively balancing model complexity and prediction accuracy. By extracting feature importance, we guide the dynamic generation of enhanced hint sets, improving optimization performance. Pruning strategies for tree count and depth enhance the interpretability of decision rules with minimal impact on accuracy. Extensive experiments demonstrate the effectiveness and efficiency of LIO. While currently focused on query plan selection, LIO's core framework based on evolutionary forests and structured query encoding has strong potential for broader applications. In the future, we aim to extend LIO to tasks such as cardinality estimation and index recommendation by adapting the input features, incorporating execution feedback, and modifying the output structure.

## ACKNOWLEDGEMENTS

This paper was partially supported by the National Key Research and Development Program of China (No. 2022YFE0199300) and the National Natural Science Foundation of China (No. 62202132).

# REFERENCES


[1] P. Akioyamen, Z. Yi, and R. Marcus. The unreasonable effectiveness of llms for query optimization. *CoRR*, abs/2411.02862, 2024.

[2] T. Chen, J. Gao, H. Chen, and Y. Tu. Loger: A learned optimizer towards generating efficient and robust query execution plans. *Proc. VLDB Endow.*, 16(7):1777–1789, 2023.

[3] B. Ding, S. Chaudhuri, J. Gehrke, and V. R. Narasayya. DSB: A decision support benchmark for workload-driven and traditional database systems. *Proc. VLDB Endow.*, 14(13):3376–3388, 2021.

[4] B. Ding, S. Das, R. Marcus, W. Wu, S. Chaudhuri, and V. R. Narasayya. Ai meets ai: Leveraging query executions to improve index recommendations. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1241–1258, 2019.

[5] F. Doshi-Velez and B. Kim. Towards a rigorous science of interpretable machine learning. *arXiv preprint*, 2017.

[6] V. Giannakouris and I. Trummer. $\lambda$-tune: Harnessing large language models for automated database system tuning. *Proc. ACM Manag. Data*, 3(1):2:1–2:26, 2025.

[7] J. Koza. On the programming of computers by means of natural selection. *Genetic programming*, 1992.

[8] B. Kroth, S. Matusevych, R. Alotaibi, Y. Zhu, A. Gruenheid, and Y. Tian. MLOS in action: Bridging the gap between experimentation and auto-tuning in the cloud. *Proc. VLDB Endow.*, 17(12):4269–4272, 2024.

[9] W. La Cava, L. Spector, and K. Danai. Epsilon-lexicase selection for regression. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, pages 741–748, 2016.

[10] J. Lao, Y. Wang, Y. Li, J. Wang, Y. Zhang, Z. Cheng, W. Chen, M. Tang, and J. Wang. Gptuner: A manual-reading database tuning system via gpt-guided bayesian optimization. *Proc. VLDB Endow.*, 17(8):1939–1952, 2024.

[11] V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann. How good are query optimizers, really? *Proc. VLDB Endow.*, 9(3):204–215, Nov. 2015.

[12] B. Liu and R. Mazumder. Forestprune: Compact depth-pruned tree ensembles. In *International Conference on Artificial Intelligence and Statistics*, pages 9417–9428. PMLR, 2023.

[13] R. Marcus, P. Negi, H. Mao, N. Tatbul, M. Alizadeh, and T. Kraska. Bao: Making learned query optimization practical. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 1275–1288, 2021.

[14] R. Marcus, P. Negi, H. Mao, C. Zhang, M. Alizadeh, T. Kraska, O. Papaemmanouil, and N. Tatbul. Neo: a learned query optimizer. *Proc. VLDB Endow.*, 12(11):1705–1718, July 2019.

[15] R. Marcus and O. Papaemmanouil. Deep reinforcement learning for join order enumeration. In *Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, pages 1–4, 2018.

[16] R. Marcus and O. Papaemmanouil. Plan-structured deep neural network models for query performance prediction. *arXiv preprint arXiv:1902.00132*, 2019.

[17] P. Negi, R. Marcus, A. Kipf, H. Mao, N. Tatbul, T. Kraska, and M. Alizadeh. Flow-loss: Learning cardinality estimates that matter. *Proc. VLDB Endow.*, 14(11):2019–2032, jul 2021.

[18] J. P. Papa, A. X. Falcao, V. H. C. De Albuquerque, and J. M. R. Tavares. Efficient supervised optimum-path forest classification for large datasets. *Pattern Recognition*, 45(1):512–520, 2012.

[19] J. P. Papa, A. X. Falcao, and C. T. Suzuki. Supervised pattern classification based on optimum-path forest. *International Journal of Imaging Systems and Technology*, 19(2):120–131, 2009.

[20] M. Pöss, B. Smith, L. Kollár, and P. Larson. Tpc-ds, taking decision support benchmarking to the next level. In M. J. Franklin, B. Moon, and A. Ailamaki, editors, *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, USA, June 3-6, 2002*, pages 582–587. ACM, 2002.

[21] C. Rudin. Stop explaining black box machine learning models for high stakes decisions and use interpretable models instead. *Nature Machine Intelligence*, 1(5):206–215, 2019.

[22] J. Sun and G. Li. An end-to-end learning-based cost estimator. *Proc. VLDB Endow.*, 13(3):307–319, 2019.

[23] R. Sun and M. Hong. Improved iteration complexity bounds of cyclic block coordinate descent for convex problems. *Advances in Neural Information Processing Systems*, 28, 2015.

[24] J. Tan, K. Zhao, R. Li, J. X. Yu, C. Piao, H. Cheng, H. Meng, D. Zhao, and Y. Rong. Can large language models be query optimizer for relational databases? *CoRR*, abs/2502.05562, 2025.

[25] I. Trummer. DB-BERT: A database tuning tool that "reads the manual". In Z. G. Ives, A. Bonifati, and A. E. Abbadi, editors, *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, pages 190–203. ACM, 2022.

[26] L. Woltmann, J. Thiessat, C. Hartmann, D. Habich, and W. Lehner. Fastgres: Making learned query optimizer hinting effective. *Proc. VLDB Endow.*, 16(11):3310–3322, 2023.

[27] Z. Wu, A. Shaikhha, R. Zhu, K. Zeng, Y. Han, and J. Zhou. Bayescard: Revitilizing bayesian frameworks for cardinality estimation. *CoRR*, abs/2012.14743, 2020.

[28] Z. Yang, W.-L. Chiang, S. Luan, G. Mittal, M. Luo, and I. Stoica. Balsa: Learning a query optimizer without expert demonstrations. In *Proceedings of the 2022 International Conference on Management of Data*, pages 931–944, 2022.

[29] X. Yu, C. Chai, G. Li, and J. Liu. Cost-based or learning-based? a hybrid query optimizer for query plan selection. *Proc. VLDB Endow.*, 15(13):3924–3936, 2022.

[30] Y. Zhao, G. Cong, J. Shi, and C. Miao. Queryformer: a tree transformer model for query plan representation. *Proc. VLDB Endow.*, 15(8):1658–1670, Apr. 2022.

[31] J. K. Zhi Kang, Gaurav, S. Y. Tan, F. Cheng, S. Sun, and B. He. Efficient deep learning pipelines for accurate cost estimations over large scale query workload. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 1014–1022, 2021.

[32] R. Zhu, W. Chen, B. Ding, X. Chen, A. Pfadler, Z. Wu, and J. Zhou. Lero: A learning-to-rank query optimizer. *Proc. VLDB Endow.*, 16(6):1466–1479, 2023.

[33] P. Zyblewski and M. Woźniak. Novel clustering-based pruning algorithms. *Pattern Analysis and Applications*, 23(3):1049–1058, 2020.