The VLDB Journal (2026) 35:21
https://doi.org/10.1007/s00778-026-00973-9

REGULAR PAPER

LEON$^+$: towards robust ML-aided query optimization

Xu Chen[1] · Ximu Zeng[1] · Yuze Wang[1] · Zibo Liang[1] · Kai Zeng[2] · Han Su[1,3] · Kai Zheng[1,3]

Received: 18 November 2024 / Revised: 12 November 2025 / Accepted: 10 March 2026

Abstract

Query optimization has long been a fundamental yet challenging topic in the database field. With the prosperity of machine learning (ML), some recent works have shown the advantages of reinforcement learning (RL) based learned query optimizer. However, they suffer from fundamental limitations due to the data-driven nature of ML. Motivated by the ML characteristics and database maturity, we propose LEON–a framework for ML-aidEd query OptimizatioN. LEON improves the expert query optimizer to self-adjust to the particular deployment by leveraging ML and the fundamental knowledge in the expert query optimizer. Different from the previous regression objective, we propose a pair-wise ranking objective and train a ranking model for plans. To help the optimizer escape the local minima and avoid failure, a ranking and uncertainty-based exploration strategy is proposed, which discovers the valuable plans to aid the optimizer. To enhance the robustness and practicality of our framework, we introduce an advanced version of the LEON framework, referred to as LEON$^+$. By dynamically adjusting the optimization space, we significantly enhance the framework's robustness against unseen workloads and drastically reduce the costs associated with exploration. We have seamlessly integrated the LEON$^+$ framework into traditional optimizers, enabling unobtrusive and automated tuning. Extensive experiments offer evidence that the proposed framework can outperform the state-of-the-art methods in terms of end-to-end latency performance, training efficiency, and stability.

Keywords Query optimization · Learned query optimizer · Cost estimation · Ranking model

1 Introduction

The query optimizer, a crucial part of database management systems (DBMS), is the most significant aspect that can affect a DBMS's performance. It aims to find the optimal query execution plans for a given SQL query before the actual execution. With the increasing complexity of DBMS, the query optimizer needs to be carefully tuned by database experts. Despite decades of research [40], query optimizer still struggles to deliver satisfactory performance and is time-consuming to maintain [20].

Background: Recently, studies on machine learning for databases (ML4DB) have attracted more and more attention and shown the superiority of boosting traditional database performance in a data-driven way [2, 21, 24, 41, 58]. In particular, reinforcement learning (RL) is applied to build a standalone query optimizer to generate query plans and demonstrates its advantages in finding competitive query plans without the help of a traditional query optimizer [18, 26, 27, 54]. However, none of the "replacement" methods have been applied in practical usage.

X. Chen, X. Zeng: These authors contributed equally to this work

✉ Han Su
hansu@uestc.edu.cn

✉ Kai Zheng
zhengkai@uestc.edu.cn

Xu Chen
xuchen@std.uestc.edu.cn

Ximu Zeng
ximuzeng@std.uestc.edu.cn

Yuze Wang
yzwang@std.uestc.edu.cn

Zibo Liang
zbliang@std.uestc.edu.cn

Kai Zeng
kai.zeng@huawei.com

[1] University of Electronic Science and Technology of China, Chengdu, China

[2] Huawei Technologies Co., Ltd, Guangdong, China

[3] School of Computer Science and Engineering, University of Electronic Science and Technology of China, Chengdu, China

Published online: 03 April 2026

Based on the lessons learned from previous work, we argue that *machine learning (ML) can hardly replace the traditional query optimizer.* We make our argument based on the fact that ML-based methods learn domain-specific knowledge in a data-driven manner. Thus, it suffers from the following two fundamental limitations:

(**L1**): ML is not an universal solution. It cannot replace the basic knowledge and axioms embedded in database systems like relational algebra, query rewriting rules and logical equivalent rules. Previous RL-based methods only solve simplified select-project-join (SPJ) queries but are unable to handle complicated situations such as subquery, where much more rules and search space should be learned. In addition, the ML model needs to learn from scratch once more transformation rules are added to the optimizer. Otherwise, it could fail due to the unknown search space. On the contrary, the traditional query optimizer is a full-fledged and general-purpose system [40] that has been studied for more than three decades. It is widely used to handle almost any situation and there are various optimizations for it.

(**L2**): ML methods suffer from the infamous cold-start problem [5]. Query optimizers are "safety-critical" systems where significant query performance regression should be avoided [8, 25]. However, the performance of ML models might fluctuate significantly during the training phase when the data is a bottleneck. Previous research on learning-based methods assumes that there is enough training data available [26, 27], which is not always the case in practice. In contrast, traditional query optimizers are known to be efficient and effective in producing reasonable execution plans in most cases. Even though some knowledge can be learned by ML, it is more reasonable for a well-established DBMS to continue utilizing existing knowledge (an expert optimizer) rather than replacing it. For example, for a low selectivity scan operator, the DBMS knows there is a high chance that the index scan will be much more efficient than the sequential scan, while the ML methods have to collect execution feedback to recognize that.

According to the aforementioned limitations, we summarize the following key design principles:

(**P1**): *ML should aid the traditional query optimizer instead of replacing it.* As mentioned in (**L1**), the traditional query optimizer is general-purpose. As a result, it is only capable of maintaining coarse-grained knowledge, such as histograms and knobs, rather than fully utilizing domain-specific knowledge, such as database instances, execution engines, and underlying data distribution, at which ML is skilled. In fact, the most imperfectness of traditional query optimizers results from the inability to perceive such domain-specific factors, leading to poor cardinality estimation, cost modeling, or knob configuration [3, 15, 20, 25, 42, 43]. In other words, ML should aid the traditional query optimizer to compensate for the aforementioned flaws.

(**P2**): *ML can utilize the prior knowledge from the traditional query optimizer to accelerate training and avoid failure.* Most previous works start training ML models from an empty or randomly filled knowledge base, which is time-consuming to surpass the traditional optimizer. As mentioned in (**L2**), expert knowledge in databases can be helpful when ML faces practical challenges. Thus, instead of learning from scratch, the ML model can start from the knowledge of the traditional query optimizer.

LEON: Based on the design principles, we propose LEON for leveraging ML to aid the query optimizer in the previous conference version [7]. Specifically, we leverage the power of ML in a ranking model to make the traditional query optimizer start from its current performance and self-adjust to a certain dataset or workload, which is beyond what a database architect or human expert can do manually. LEON combines the advance of learning-based methods and expert knowledge in the query optimizer. Even if the ranking model fails, it can easily fall back to the traditional query optimizer.

Specifically, we leverage the basic knowledge including rewriting rules, transformation rules, and standard search strategy from the DBMS. We design a mixed cost estimation combining both the expert cost model and the ranking model to guide the plan search process. LEON takes the cost model as the initial cost estimation and uses a ranking model for cost calibration, to make the cost model more consistent with the user-defined goal (e.g., latency). The ranking model will only *calibrate* the erroneous cost estimation from the cost model based on execution history or existing query logs instead of learning from scratch. For example, when LEON finds the cost model over-estimates a sub-plan during plan search, it will automatically calibrate the cost to a lower value compared to other plans. In this way, the cost model will guide the query optimizer to find the best execution plan. In Sec. 7.6, we find that leveraging expert knowledge has a strong inductive bias and a restriction when the learning component is not effective, which gives the learned optimizer's performance a great lower bound guarantee.

While this might sound like a straightforward solution, it is not. In fact, as we will show, learning the cost model is a non-trivial task. There are two key concepts serving as learning objectives:

(**O1**): *The query optimization problem is essentially a ranking problem instead of a regression problem.* Existing works typically treat cost estimation as a supervised regression problem [42, 43]. However, the widely used accuracy metric for cost estimation cannot reflect a method's end-to-end query performance [33], which prevents them from deployment in DBMSes. Intuitively, regardless of the absolute value, the plan decision is only based on the ranking of the candi-

date execution plans. No matter how terrible the tail plans are, the best plans can be chosen from the top-k candidate plans. Actually, the *Learning to Rank* (LTR) problem has been studied extensively in recommendation systems [9, 56]. It has been shown that pair-wise ranking approaches are more widely used compared to point-wise approaches (i.e., learning absolute values). In this paper, we formalize cost estimation as a contextual pair-wise ranking problem. We train the ranking model to learn the relative order between two plans regardless of the absolute value of cost estimation.

(**O2**): *Based on* (***O1***), *we consider ranking and quantifying the uncertainty of the ranking model to explore valuable plan space to enhance model performance.* Exploitation versus exploration is a critical topic in the ML community, which also applies to the learning-based optimizers [25, 26, 50]. We design a robust plan exploration strategy to strike a great balance. There are two important factors: (1) To explore more efficiently, our first insight is that a plan with a higher ranking position should be explored with a higher probability since the optimizer inherently cares more about the higher-ranked plan than the lower one. (2) To explore more effectively, our second insight is that the learned optimizer should correct its errors that lead to sub-optimal query plans. We extend the ranking model to generate cost calibration and corresponding *uncertainty* for that calibration simultaneously. We dig deep into them and find that the erroneous execution plans have a positive correlation to the uncertainty of the ranking model. Thus, we define two exploration criteria— top-k ranking and uncertainty to solicit the potential plans. Finally, selected samples will be executed to collect execution feedback for training.

Based on the design principles, the initial conference version [7] of this study has demonstrated its effectiveness in aiding the query optimizer. However, LEON still faces potential problems in the following two aspects. (1) Potential performance regression: LEON aids the query optimizer by learning from historical query execution feedback. Given the extensive plan space for queries, there is a potential performance regression when LEON operates on a large and unseen plan space. For example, when handling a new query, LEON may extend the plan space to unseen plans with different predicates or join conditions, thus leading to incorrect predictions. (2) Exploration overhead: another limitation is that LEON potentially incurs exploration overhead. Specifically, to identify valuable plans for improving cost calibrations, LEON executes additional superior plans. As a result, there is a lack of a control mechanism to manage the overhead of exploration.

LEON$^+$: To address the issues mentioned above, we introduce a robust version of LEON called LEON$^+$. Unlike the full-level optimization (refer to Sec. 2.4) in LEON with the aim of maximizing performance gains, we design LEON$^+$ to provide a *safe* (i.e., with minimal performance regression) and *efficient* (i.e., with minimal training overhead) ML-aided query optimization framework. Our core assumption is that enhancing overall query optimization can be achieved by focusing the scope of the ranking model on a key *subset* of plan spaces. Building on this foundation, we propose four key improvements that significantly enhance LEON's performance. (1) To mitigate potential performance regressions, we introduce the concept of an *space* in LEON$^+$. Defined as a specific subset of the entire plan space, the optimization space selectively enhances the traditional query optimizer's performance. The optimization space is designed to maximize performance gains while minimizing regression risks, as discussed in Section 2.4. (2) As the search space grows exponentially with the complexity of queries [20], we propose an efficient top-down exploration strategy to manage the optimization space dynamically using accumulated historical knowledge in Section 5.1. Such a strategy can reduce exploration overhead significantly while preserving performance gains. (3) To enhance the robustness of the ranking model, we improved the generalization of LEON$^+$ by introducing a lightweight validation model in Section 4.3. This validation model acts as a gatekeeper to prevent the selection of plans that would lead to performance deterioration. (4) We further implement a seamless integration of LEON$^+$ into PostgreSQL without disrupting standard database services, demonstrating its practicality in Section 6. We have open-sourced our code [1], which contributes to research on the integration of AI with databases.

In summary, our contributions are as follows.

1. We present LEON$^+$, a safe and efficient ML-aided learning framework for the expert query optimizer. LEON$^+$ integrates ML models deeply into traditional optimizers, combining both ML and expert knowledge.
2. We propose a contextual pairwise ranking objective for query optimization, which aims to help the ML-aided optimizer make better decisions.
3. We introduce an optimization space to limit the model calibration scope within the query optimizer and dynamically manage the space, enhancing efficiency and efficacy without compromising performance.
4. We enhance the generalization of LEON$^+$ by blocking deteriorated plans with a validation model significantly.
5. We further integrate LEON$^+$ into PostgreSQL to leverage substantial plan space without affecting normal database services.

[1] https://github.com/Thisislegit/LeonProject

2 Background and motivation

In this section, we first describe the traditional query optimizer. Then we analyze how ML methods tackle query optimization and compare them to traditional query optimizers. We summarize the existing learning-based query optimization methods and corresponding characteristics. In general, we can categorize them into two classes: *ML-replaced* and *ML-aided*. Finally, we discuss different types of granularity for ML-aided components.

2.1 Standard query optimizer

The basic paradigm of a traditional query optimizer is to *enumerate* candidate plans and then search for the optimal one among them with a *cost model*. Dynamic programming (DP) is utilized as the core search strategy [13, 14, 40]. The DP enumeration module is based on the optimality principle and memorization. Optimality principle: DP decomposes the global optimal solution into iterations for the local optimal solution. To illustrate, given a complete logical expression (query) Q, the equivalent set S is defined by a combination of the logical expression q (q can be a partial expression of Q) and physical property ω, which is denoted by $S = (q, \omega)$. DP continuously enumerates larger S into physical plans p and obtains the optimal solution until S is as large as Q. We denote by $\mathcal{C}(S)$ the set of rule–enumerated candidate *physical* plans for the equivalent set S. Memorization: A look-up table keeps track of the optimal plan for explored equivalent sets. The suboptimization decisions (subplans) in the look-up table can be used in the complete plan.

Here we describe an overview of the bottom-up search engine. The search engine finds possible execution plans for a query by successively iterating on the number of relations joined so far. The input to the search engine is a set of base relations (denoted by $R(q)$). Then for each level i: (1) **Plan Enumeration**: The plans containing i base relations for the same equivalent set S will be generated based on the combination of former optimal plans saved in a look-up table. (2) **Cost Computation**: The statistic for every plan will be derived to compute the cost by a cost model. (3) **Cost Comparison and Memorization**: The optimal plan and corresponding cost from the set of candidate plans $\mathcal{C}(S)$ are chosen and memorized in the look-up table. (4) If $i = |R(q)|$, the search process finishes, and the optimal plan is returned. Otherwise, go back to step (1).

To discover the optimal execution plan, two conditions must hold: (i) *complete and correct plan enumeration*: the enumeration module should generate the entire legal plan space for each equivalent set $S = (q, \omega)$ (no admissible implementation is omitted) and every generated plan must be semantically equivalent to q while satisfying ω; and (ii) *effective costing*: the optimizer must assign sufficiently accurate costs to candidates in $\mathcal{C}(S)$ to rank them reliably. When both are met, the DP search returns the globally optimal plan.

2.2 Why learn what we already know?

Recent works leverage reinforcement learning (RL) techniques to learn an end-to-end query optimizer to replace the traditional query optimizer [18, 26, 27, 50, 54]. We call them ML-replaced methods. In this subsection, we compare these methods with the standard query optimizer in two crucial components in detail. We find that the fundamental knowledge and axioms in traditional query optimizers cannot be and need not be learned by ML models.

Plan Enumeration. The transformation rules exhibit the basic knowledge of query optimization, which should be preserved. Transformation rules, specifying equivalence transformations for logical expressions and physical implementations, represent the knowledge of algebraic law for plan enumeration in an equivalent set. For example, the transformation rules can unnest an IN/EXISTS subquery to a semi-join, which expands search space. A subquery can also be pushed up to be evaluated in advance, so that it can appear in the earlier steps of the overall execution plan, thereby obtaining a better execution plan. On the contrary, an ML model, learned in a data-driven fashion, can hardly reason out such complex rules and patterns. As a result, existing ML-replaced methods enumerate plans based on fixed rules: 1) basic associative and commutative laws for joins; 2) pushing projection and predicates to leaf nodes for scan and filtering. This can lead to an incomplete plan enumeration and potentially suboptimal query plans.

In addition, modern query optimizers have great extensibility. They can be extended with new operators, cost models, properties, and rules. As DBMSes advance, more rules and implementation algorithms can be added as the basic knowledge of a DBMS since components are independently modularized. For comparison, the learned query optimizer has to learn from the exponential growth equivalent sets in a trial-and-error manner.

Cost Model. After enumerating various candidate plans, the optimizer selects an optimal plan and then prunes other plans in an equivalent set. The traditional query optimizer uses a heuristic-based cost model $C(\cdot)$ to estimate the execution cost of plan p denoted as $C : p \rightarrow$ cost. The expert-implemented cost model also contains a large amount of knowledge. The cost estimation is based on knowledge of multiple factors, such as the cost of I/O count, CPU time, and coarse-grained statistics for data distribution, etc. The cost model, although not accurate, is an off-the-shelf score function with decent performance. More importantly, it is robust to the dynamic workload and data shifting, which is what ML models suffer from. Recent RL-based methods all try to learn from existing cost models. For example, Neo

Table 1 A summary of learning-based query optimization methods, which can be categorized into four categories: (1) RL-based end-to-end query optimizer, (2) learned knob/hint tuner (3) learned cardinality estimation (CardEst), (4) learned cost model (CostEst)

Characteristics	Methods				
	ML-Replaced	ML-Aided			
	RL-based [18, 26, 27, 50, 54]	KnobTuner [3, 22, 25, 55, 57]	CardEst [15, 26, 29, 30, 34, 49, 51]	CostEst [28, 42, 43]	LEON$^+$ **(Ours)**
DB Knowledge	–	✓	✓	✓	✓
Black/White-Box	Black-Box	Black-Box	White-Box	White-Box	White-Box
Ranking/Regression	Regression	Regression	Regression	Regression	Ranking
Exploration	✓	✓	–	–	✓
DB Integration	–	Deep	Shallow	Shallow	Deep

[26] collects expertise experience from a traditional query optimizer. RTOS [54] pre-trains the ML model by cost as supervision. The cost model has been demonstrated to help the ML model. However, learning from the existing knowledge and replacing it can be a waste of time. As certain database applications are critical to organizational mission and operations, maintaining a worst-case optimal cost model is necessary for DBMS vendors. Although it is desirable to replace this model, it is currently impractical. Therefore, for established database systems, it is unwise to abandon all existing work. Instead, our approach is to supplement the current model with a new technique.

Given the shortcomings of the ML-replaced methods, in the next subsection, we analyze how different learning-based approaches aid an expert query optimizer.

2.3 What should we learn to aid query optimizer?

The ML-aided approaches build ML models on top of traditional query optimizers to enhance optimizer performance. The ML-aided approaches can be categorized into two classes: *black-box* and *white-box*. The white-box methods influence query optimizer behavior for sub-optimization (e.g., changing a join algorithm to another for a specific join operator), while black-box methods can only influence the entire plan. In this subsection, we analyze those two types of methods and give our opinions as shown in Table 1.

Black-box Methods. The black-box methods include knob/hint Tuner. DBMSes provide some knobs for DBAs to fine-tune their performance for specific applications. Learning-based knob tuning [3, 22, 55] uses RL to fine-tune the parameters (e.g., working memory). Specifically, many DBMSes provide hint sets for DBAs to fine-tune query optimizer behavior. Bao [25], different from the previous methods, steers an expert query optimizer by tuning hint sets (e.g., disabling nested loop join) for each query. The hint choices depend on the latency prediction from Bao's predictive model. However, black-box methods have a fundamental limitation: they only have coarse-grained optimization choices (enabling/disabling operators for the entire plan). For example, a subplan is optimal with a nested a nested-loop join may be suboptimal in another, unrelated equivalent set. On the contrary, by manipulating the cost model, the potential set of plans that can be generated extends strictly beyond that of Bao's.

White-box Methods. The white-box methods include learned cardinality estimation (CardEst) [15, 26, 29, 30, 34, 49, 51] and learned cost estimation (CostEst) [28, 42, 43]. They aim to learn a parameterized model to steer the expert cost model or the cardinality estimator. However, they have some drawbacks that are deeply related to query optimization.

(1) Query optimizer only cares about the ranking of plans. Previous work normally leverages an ML model to predict the latency of a plan. However, learning the absolute value is hard for an ML model considering the latency can be vastly different between two plans. Instead, the ML model only needs to learn the relative relationship between two plans, which relaxes the requirement for ML model training. Moreover, the ML model needs to learn more about higher-ranked plans instead of less favorable plans. For a bad plan, the exact performance prediction is uncritical.

(2) Plan ranking is only meaningful for the candidate plans within the same equivalent set i.e., logically equivalent plans with the same physical property. Previous methods learn plans without the equivalent set restriction. Instead, they should focus on comparing plans within the same equivalent set, which could reduce unnecessary plan comparisons.

(3) Finding better plans has to jump out of the existing experience, which shows the necessity of plan exploration. Only relying on the current optimal plans can make optimization performance stuck at local minima. Instead, a query optimizer can only progress if it explores extra potentially good execution plans. However, previous white-box methods do not pay attention to exploration but only train the ML models on a static workload. Once the data shifts, the ML model can be fragile.

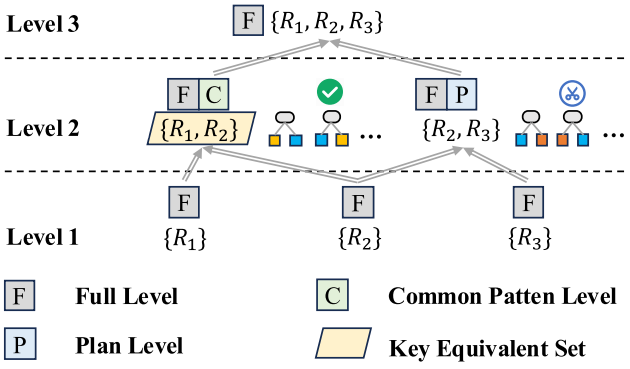

Fig. 1 Different granularity for ML-aided optimizer

In summary, none of the existing approaches can integrate deeply into the expert query optimizer and influence the optimization decisions effectively. However, each method has its advantages, which make us decide to go the way of a white-box ML-aided query optimizer. To train ML models, we should first make the expert optimizer an initialization. Then we need to make the ranking an objective and explore potentially better optimization decisions. In this way, we can fully leverage the knowledge in the expert query optimizer. The ML-aided optimizer starts from the current expert performance and self-adjusts towards the deployed database instance.

2.4 Granularity aided by ML

ML-aided optimizers all follow the same high-level paradigm: learn from historical workloads and *inject* that knowledge back into a traditional cost-based optimizer to bias plan search. What differs is the *granularity* at which ML intervenes in the plan space. As summarized in Fig. 1, prior systems fall into three categories.

(1) Plan level (coarse global control). Knob tuners generally operate at this level. ML biases search via coarse global knobs (e.g., disabling nested-loop joins), uniformly trimming the entire candidate space, as shown in the blue block of Fig. 1. This delivers highly efficient learning but breaks plan-space completeness: some admissible implementations are never enumerated. Thus, this category offers limited effectiveness on heterogeneous workloads.

(2) Common-pattern level (localized control). ML acts only on subspaces associated with recurring patterns such as query templates. Here, the decision is conditional on the local structure (e.g., an equivalent set that matches a template), enabling targeted guidance while leaving unrelated regions to the expert optimizer. Many industrial deployments adopt this middle ground for a better efficiency–effectiveness balance [16, 42, 47].

(3) Full level (global fine-grained control). ML participates throughout the optimizer, potentially re-scoring or re-ordering candidates across all equivalent sets. This maximizes potential effectiveness but raises engineering complexity and latency/overhead risk.

Table 2 Summary of Notation

Notation	Definition
$\mathcal{W}$	A workload
Q	A complete query
q	A partial logical expression of Q
p	A plan corresponding to q
ω	Physical property
$S = (q, \omega)$	Equivalent set of q and ω
$\mathcal{C}(S)$	Set of candidate plans from S
$R(q)$	A set of base relations
$C(\cdot)$	Heuristic-based cost model
S_k	Key equivalent sets
T	Template queries
A	Optimization space
$M^R(\cdot,\cdot)$	The ML plan ranking model
$M^V(\cdot,\cdot)$	The validation model
k_{time}	Exploratin time budget
$L(\cdot)$	Execution Latency
E	Experience pool

Our previous work, LEON [7], chose full-level optimization. By leveraging a comprehensive approach, LEON aimed to maximize performance gains through nuanced adjustments. In this work, we shift our focus to optimizing the framework robustly and efficiently through common pattern level optimization. The optimization based on the common pattern level is motivated by reasons: (1) We find that optimizing a *key* group of common patterns can significantly aid the overall query optimization, which is also demonstrated in some recent works [19, 47]. (2) Workloads are highly repetitive in real-world situations [32, 38, 48], leading to significant opportunities for common pattern optimization. Based on these motivations, LEON$^+$ carefully organizes the learned knowledge into structured common pattern without sacrificing the integrity of the plan space. The common patterns are defined as **template queries** in Sec. 3.1, which can also evolve dynamically to accommodate unseen workloads.

3 Framework overview

In this section, we briefly introduce a set of preliminary definitions in the ML-aided optimizer and then give an overview of our framework. Table 2 lists the major notations used in the paper.

3.1 Preliminaries

An ML-aided query optimizer is a database query optimization component that uses ML to enhance query plan selection and cost estimation. Here we provide the objective of the ML-aided optimizer.

Definition 1 (*Objective*) Given historical workload traces with execution feedback $\mathcal{W}_{\text{hist}} = \{(Q_i, p_i, L_i(p_i))\}_{i=1}^{t}$, where each Q_i is a query, p_i is the chosen plan, and $L_i(\cdot)$ is the latency, the optimizer continuously updates its learned component to complement the traditional optimizer. The objective is, conditioned on the available history $\mathcal{W}_{\text{hist}}$, to select plans for forthcoming queries $\mathcal{W}_{\text{fut}} = \{Q_{t+1}, \ldots, Q_{t+T}\}$ so as to minimize end-to-end latency $L(p)$ on the future workload, while keeping the likelihood and magnitude of regressions relative to the expert baseline p^O as small as possible. As feedback accumulates, the optimizer should generalize to previously unseen queries and adapt to workload evolution without degrading performance.

The standard plan search algorithm decomposes the plan space into subset problems (i.e., equivalent sets). Here, we identify the key equivalent set to achieve the aforementioned objective of optimal performance with minimal overhead.

Definition 2 (*Key Equivalent Set*) We define a key set of equivalent sets ($S_k \in Q$) as any subset of equivalent sets (S) of a query Q that meets two criteria: (1) Only applying the ML-aided cost estimation of S_k while using the traditional cost estimation to the remaining equivalent sets, should generate an identical optimal final plan as obtained by using the precise costs for the entire equivalent sets S. (2) S_k is the smallest subset that enables the optimal final plan, with any reduction in S_k resulting in a suboptimal plan.

The above definition suggests a hypothesis following [19]: by carefully identifying S_k and calibrating the plan selection in S_k, it is possible to efficiently aid the cost-based optimizer in generating an optimal plan. Thus, this motivates us to distill the knowledge learned from $\mathcal{W}_{\text{hist}}$ into a finite, well-scoped region (*optimization space*) and to manage it in a structured, template-based form (*template queries*). In Figs. 1 and 2, we use the yellow diamond to denote the key equivalent set.

Definition 3 (*optimization Space*) The optimization space A is defined as the influence scope of the ML component within a traditional query optimizer, where the ML component can enhance the query optimization process.

Ideally, the optimization space should encompass all key equivalent sets, ensuring that the ML component can optimize the most impactful portions of the query plan space. As discussed in Section 2.4, to ensure both effectiveness and efficiency, we aim to identify common patterns within the historical workload. We introduce template queries that capture the shared characteristics across various queries, which we define as follows.

Definition 4 (*Template Query*) A template query, denoted as T, is an abstract representation of a set of queries that share an identical logical structure but differ in specific constant or parameter values.

We consider an equivalent set S to belong to a template query T when they have the same query structure and logical operations, differing only in the constant values denoted as $S \subseteq T$. We regard template queries as the fundamental units within the optimization space since they approximate the key equivalent sets, denoted as $A = \{T_1, T_2, \ldots, T_n\}$. Consequently, if an equivalent set S is a subset of any template query within the optimization space A, then S is also a subset of the optimization space A, denoted as $S \subseteq A$. Additionally, we manage the optimization space dynamically to explore the key equivalent sets and adapt to changes in the workload, ensuring that the ML component continues to provide effective optimizations in response to evolving query patterns.

3.2 ML-aided query optimizer

The standard search process is described in Section 2.1. ML component influences a standard query optimizer's optimization decision in the following aspects (green boxes in Fig. 2). For the cost computation and cost comparison step in Section 2.1, LEON$^+$ introduces two extra steps, including *plan ranking* and *validation*.

Plan Ranking. As shown in Fig. 2, LEON$^+$ maintains an optimization space consisting of the template queries (e.g., T_1 and T_2). Optimization space is presented with parallelograms, while those equivalent sets not in optimization space are depicted with ovals. For the cost computation and cost comparison step, LEON$^+$ first identifies whether the current equivalent set S is a subset of any template query within the optimization space A. If $S \nsubseteq A$, LEON$^+$ utilizes expert cost model to make cost computation. If $S \subseteq A$, instead of relying on the expert cost model, LEON$^+$ learns a score function as follows.

$$M^R : (LF, PF) \rightarrow (\text{score, uncertainty}) \tag{1}$$

The ranking model M^R maps a (logical feature (LF), physical feature (PF)) pair to a scalar value (score) to rank plans from the same equivalent set S. $LF = (Q, q)$ is defined by complete query Q and current logical expression q and $PF = (\omega, p)$ is defined by physical property ω and current plan p. Thus, the ranking position of a plan in the equivalent set S is reordered by the score. The optimal plan in the equivalent set is chosen by selecting the plan with the

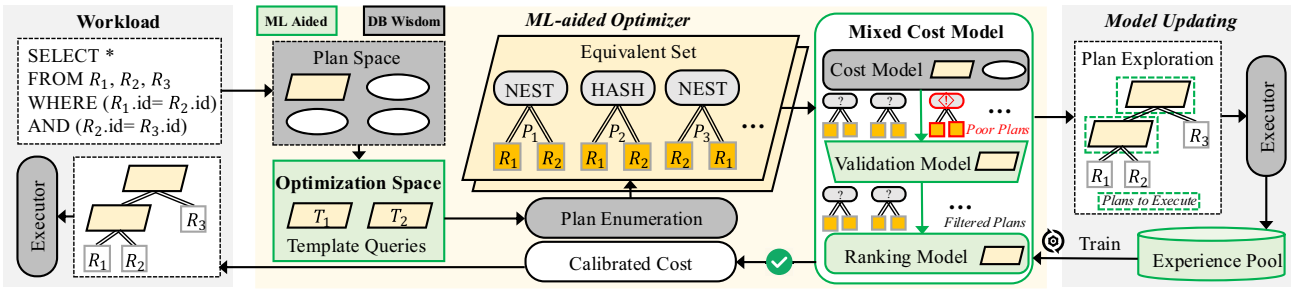

Fig. 2 An overview of the ML-aided query optimizer. The optimizer maintains the DB knowledge and self-adjusts towards a database instance by ML models

lowest estimated score from M^R. Note that M^R also computes the uncertainty of the score (details in Section 5.2). The uncertainty indicates how confident is the model M^R to the predicted ranking position, which is aimed at plan exploration. M^R can easily disable the exploration mode during the online inference.

Compared to the expert cost model, the absolute value from the score function has no semantic meaning since it only learns the relative ranking position for plans in intra-equivalent sets (formally defined as *context* in Section 4.1). In addition, the learned score function is non-trivially built. It incorporates the expert cost model as an initialization ($M^R(LF, PF) \approx C(p)$ at the beginning). This approach mitigates the cold start problem as it avoids the need to collect a large amount of data when the ranking model is unstable and reduces the risk of unexpected performance regression during the learning process. We call M^R_θ a *mixed cost model*. Furthermore, it leverages collected execution feedback to make the expert cost model tailored to the target database instance. If the optimal ranking can be ordered by the score function, the optimal plans can be searched by the query optimizer.

Validation. We further introduce a lightweight validation model M^V to enhance the generalization of LEON$^+$ (details in Section 4.3). Consisting with the usage of the ranking model, the validation model M^V is also used in the optimization space. Specifically, M^V is a binary classifier:

$$M^V : (p, p^O) \rightarrow \hat{y} \tag{2}$$

It compares an encoded plan p with the encoded optimal plan p^O chosen by the traditional query optimizer. The validation model predicts whether the p is inferior to p^O. The output $\hat{y} = 0$ means the latency of plan p is better than plan p^O, while $\hat{y} = 1$ infers the opposite. Among an equivalent set, the top re-ranked plan that passes the validation model is selected (i.e., when $M^V(p, p^O) = 0$).

LEON$^+$ trains a neural network with parameter θ to approximate the optimal score function M^R_θ and M^V_θ. The inputs to the neural networks of M^R_θ are encoded as logical and physical feature vectors: The logical feature vector encodes information in Q and q. The physical feature vector encodes information in ω and p (details in Section 4.2).

3.3 ML-aided optimizer updating

For updating the ML-aided optimizer, LEON$^+$ collects experience and uses it to train ML models. Specifically, in every iteration, LEON$^+$ leverages the current ML-aided query optimizer to search plans for the training workload. During the plan search, LEON$^+$ collects extra experience. After the plan search, LEON$^+$ trains ML models with collected experience. The following three steps, (a) management of template query (details in Section 5.1), (b) experience collection (details in Section 5.2), and (c) model training (details in Section 5.3) are alternate until the predefined stopping condition is met. The workflow of model updating is illustrated on the right side of Fig. 2.

Management of Template Query. To maximize the performance with minimal overhead, LEON$^+$ adaptively manages the optimization space A. We limit the ML usage by constraining optimization space for both inference and training. We design a top-down exploration strategy to analyze historical query plans and extract the most valuable common patterns (i.e., template queries) from plans. Even with dynamic workloads and data, LEON$^+$ is capable of making timely adjustments. For example, on the right side of Fig. 2, only the top two queries (i.e., queries that are shown in parallelograms) are among the T and will be further used in the experience collection.

Experience Collection. LEON$^+$ maintains an *experience pool* $E = \{(q, Q, p, \omega, C(p), L(p))\}$ to collect execution feedback including logical expression q, complete query Q, plan p corresponding to q, physical property ω, cost $C(p)$ and latency $L(p)$. What experience to collect is a non-trivial problem. For each query in the training workload, LEON$^+$ uses a standard plan enumerator in the query optimizer. Specifically, LEON$^+$ uses an ML-aided query optimizer to search plans. For every equivalent set, LEON$^+$ has a plan exploration strategy to pick valuable (sub)plans. The plan exploration is based on two criteria: *ranking* and *uncertainty*

derived from current M_θ^R to discover potentially better plans. The selected plans will be used to collect their execution feedback saved in E.

Model Training. Model training aims to let the ranking model learn from the collected experience E. In the beginning, LEON⁺ initializes the mixed cost model M_θ^R from the expert cost model. LEON⁺ borrows ideas from the recommendation system and formalizes query optimization as a contextual pair-wise plan ranking problem to train the model tailored to the goal (details in Section 4.1). For every iteration, LEON⁺ picks several batches of plan pairs (p_1, p_2) in experience pool E under certain contexts for the ranking model. Then, LEON⁺ trains the ranking model M_θ^R and the validation model M_θ^V with standard supervised learning fashion by our proposed contextual ranking objective and safety regularization serving as the loss function. The two models are trained on the collected experience iteratively to approximate the optimal score function.

4 ML-aided query optimizer

In this section, we first formalize the query optimization problem as a pair-wise classification problem in Section 4.1. Then we describe how to use the ranking model and the validation model in LEON⁺.

4.1 Problem formulation

The goal of query optimization is to pick the best query execution plan regarding latency. The previous learning-based method uses a ranking model to predict the plan performance in a supervised regression manner, i.e., learning the exact latency. However, in practice, such a learning objective has significant prediction errors [1].

Instead, what we need is the correct order of candidate plans. Thus, we formalize query optimization as a contextual plan ranking problem.

Definition 5 (*Contextual Plan Ranking*) Given the context defined by restrictions from three aspects: complete query Q, the logical expression q, and physical property ω, give order to enumerated physical plans from the same context. The order complies with the relative position of the target optimization goal (e.g., latency) without actually executing the plans.

Intuitively, the plan ranking problem is relatively easier than the supervised regression problem. Accurate latency prediction of a plan implies the correct ranking, while correct ranking does not need accurate prediction for a fixed value. Note that our search strategy is based on DP, we mainly care about the plan ranking in the same *context*. For intra-equivalent set optimization, context is defined as restrictions on the same Q, q, and ω. The context restriction helps LEON⁺ reduce unnecessary plan comparison.

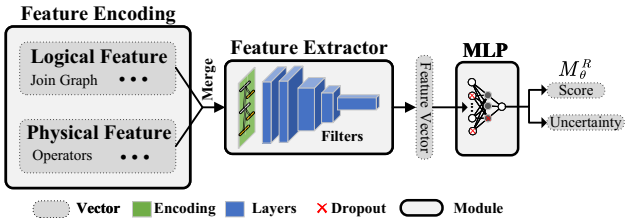

Fig. 3 Ranking model architecture

Plan ranking naturally has transitivity property, i.e., given the score function $M^R, \forall p_1, p_2, p_3$ enumerated from the same context: $M^R(p_1) > M^R(p_2) \cap M^R(p_2) > M^R(p_3) \rightarrow M^R(p_1) > M^R(p_3)$, where $M(\cdot)^R$ denotes a score function in general and we omit the same context input. Thus, we can formalize a plan ranking problem to a pair-wise ranking/classification problem.

Definition 6 (*Contextual Pair-wise Classification*) Given a pair of physical plans $\forall p_1, p_2$ derived from the same context, predict which plan has higher ordering according to the contextual plan ranking without actually executing the plans.

Learning to Rank (LTR) has been studied in the recommendation system. It has been shown that forecasting relative order is more closely related to the nature of ranking than predicting absolute value so pairwise techniques perform better in practice than pointwise approaches [9, 56]. In addition, the pairwise classification formulation is a more feasible way to train the ranking model as we only need two labels to train the model instead of the whole ranking list.

4.2 Ranking model

Network of Ranking Model. Here we describe the details of how to build appropriate ML models. The model architecture is shown in Fig. 3. The input to the feature extractor contains two aspects: logical features and physical features. The logical features are encoded by a one-hot vector to represent logical properties including output cardinality and join graph from (sub)query q and complete query Q. The physical feature maintains a tree structure to represent the plan tree p and physical property ω, where every tree node is represented by a one-hot vector indicating the physical operators and sort order. The logical one-hot encoding then merges with every tree node vector as the final encoding. Pattern matching for plan trees is commonly used and empirically verified by previous works [25, 26, 50]. LEON⁺ builds a ranking model, M_θ^R, to learn such patterns. The choice of feature extractor is not our contribution and is orthogonal to techniques of LEON⁺. Users can use other network architectures. In our implementation, we use tree convolution networks including

convolution and pooling operations in Neo [26] (denoted as filters in Fig. 3.). After the feature extractor, there are two output heads. Each head is a multilayer perceptron (MLP), which takes the feature vector as the input to predict the desired outputs. We defer the details of pairwise training to Section 5.3.

Usage of Ranking Model. LEON$^+$ trains a ranking model M_θ^R to influence the optimization decisions within an equivalent set. Based on Definition 6, the ranking model training is supervised by the immediate latency performance of plan pairs from the same equivalent set S. However, learning the parameterized cost model M_θ^R from scratch can be hard considering M_θ^R has to evaluate a large number of plans, especially when training data is a bottleneck. To this end, we propose to represent a mixed cost model M_θ^R by applying a parameterized calibration function $g_\theta(\cdot,\cdot)$ to the traditional cost estimation $C(\cdot)$:

$$M_\theta^R(LF, PF) = g_\theta(LF, PF)C(p) \tag{3}$$

where g_θ maps logical and physical features to a calibration ratio, and the $g_\theta(LF, PF)C(p)$ is the $score$ for plan ranking. In the beginning, the classification ratio will be initialized close to one ($g_\theta \approx 1$) as a much simpler initialization for practice. For the uncertainty measurement, we add dropout layers into the MLP of M_θ^R to introduce randomness (red cross in Fig. 3). Thus, M_θ^R can measure uncertainty based on multiple predictions. The rationale will be illustrated in Section 5.2.

4.3 Validation model

Motivation of Validation Model. A well-known problem in machine learning is the trade-off between bias and variance in a model. For example, large and deep neural networks (e.g., tree CNNs) tend to overfit historical data and are less effective at handling unseen data, while smaller networks often generalize better but may underfit. Therefore, especially in the context of continuous learning and the integration of new domain knowledge, the plans generated from a distinct equivalent set may lead to detrimental effects on the generalization ability of the ranking model and lead to catastrophic forgetting [12]. To address this, we employ a lightweight validation model to enhance generalization. This strategy aligns with principles from ensemble learning and model stacking, where combining models of different complexities leads to better generalization [11]. With only 0.1% of the parameters of the ranking model, the validation model is trained concurrently but on a simpler task, serving as a gatekeeper to block deteriorated plans produced by the ranking model. In the task of query optimization, we find that while it is challenging for the validation model to select optimal plans, it is quite effective at identifying catastrophic ones, which is crucial for ensuring robustness.

Usage of Validation Model. The validation model M^V predicts whether a candidate plan p will *regress* relative to the expert baseline p^O selected by the traditional optimizer from the same context. We cast this as a margin-based binary classification on plan pairs. Let $\Delta_{\text{rel}}(p, p^O) \triangleq \frac{L(p)-L(p^O)}{L(p^O)}$, be the normalized relative delta in latency.

Given a tolerance $\alpha > 0$, we assign the supervision label

$$y(p, p^O) = \begin{cases} 1, & \text{if } \Delta_{\text{rel}}(p, p^O) > \alpha \quad \text{(regression)} \\ 0, & \text{if } \Delta_{\text{rel}}(p, p^O) < -\alpha \quad \text{(improvement)}. \end{cases}$$

We encode each plan with a 70-dimensional feature vector $\phi(\cdot)$ that summarizes operator-level statistics (as in [8]), discarding tree structure. For each operator, we utilize several key features: the estimated cost of executing the current operator; the estimated number of output rows produced by the operator; the estimated byte size of the data it processes; the estimated number of rows processed; the estimated byte size handled by the operator's child components; weighted sums for the estimated rows and byte sizes at the leaf nodes of the execution plan. Then we feed the *difference* vector $\mathbf{x}(p, p^O) = \phi(p) - \phi(p^O) \in \mathbb{R}^{70}$ to a lightweight MLP classifier: $s(p, p^O) = M^V(\mathbf{x}(p, p^O)) \in [0, 1]$, interpreted as the predicted probability of regression. Intuitively, $s \approx 1$ indicates strong evidence that p is *worse* than p^O (by at least α), $s \approx 0$ indicates that p *improves* over p^O (by at least α), and $s \approx 0.5$ denotes near-equivalence. Filtered survivors are then forwarded to the ranking model on a reduced candidate set, which improves generalization while lowering both inference and training cost.

We accept a candidate plan p only if the model predicts *non-regression*, i.e.,

$$\text{Accept}(p, p^O) \iff M^V(\mathbf{x}(p, p^O))(p, p^O) \le \tau_s.$$

The *degree* of the gate is controlled by two simple knobs: (i) adjusting the decision cutoff τ_s on M^V (using a value *below* 0.5 ($\tau_s < 0.5$) makes the gate more conservative; using a value *above* 0.5 ($\tau_s > 0.5$) makes it more aggressive), and (ii) tuning the label tolerance α in $y(p, p^O)$ (which widens or narrows the near-tie band, consequently pushing $s(p, p^O)$ toward 0.5 for borderline pairs and yielding a more conservative or aggressive operating point, respectively).

5 ML-aided optimizer updating

Model updating consists of three steps: management of affection space, experience collection, and model training. The ranking model and the validation model are trained itera-

Algorithm 1 Evolving Optimization Space with Budgeted Exploration

1: **Input:** $\mathcal{W}_{\text{hist}} = \{(Q_i, p_i, L_i(p_i))\}_{i=1}^{t}$; ranking model M^R; per-set cap k_{plan}; global time budget k_{time}
2: **Output:** T, E
3: Init $T = \emptyset$ for top-down exploration
4: **for** query Q in experience E **do**
5: Extract sub-queries $q_1, q_2, \ldots, q_n$ and sub-plans $p_{q_1}, p_{q_2}, \ldots, p_{q_n}$
6: Include top-layer plan as template $T \leftarrow T \cup \{p\}$
7: **if** $|T| > \max_T$ **then**
8: Replace the template in T with the lowest improvement runtime
9: **end if**
10: **end for**
11: $B \leftarrow k_{\text{time}}$
12: **for** equivalent set S in optimization space **do**
13: **if** IDLEWINDOW() **then**
14: Let $\mathcal{C}(S)$ be candidates; compute uncertainty $u(LF, PF)$ for $p \in \mathcal{C}(S)$
15: $P_S \leftarrow \text{sort}(\mathcal{C}(S),\ \text{key} = u(LF, PF),\ \text{descending} = \text{true})$
16: **for** $p \in P_S$ **do**
17: **if** $B \leq 0$ **then break**
18: Execute p (log $T(Q, p)$ into E; $B \leftarrow B - T(Q, p)$
19: **end for**
20: **end if**; **if** $B \leq 0$ **then break**
21: **end for**
22: **return** T, E

tively. For every iteration, the current ML-aided optimizer collects execution feedback to experience pool E with a top-down exploration strategy. The ranking model is trained by learning from the pairwise samples in the experience pool to improve current ML-aided optimizer performance. We illustrate the process of optimization space management and experience collection in Algo. 1.

5.1 Optimization space management

The objective of optimization space management is to limit ML resource usage by constraining the optimization space for both inference and training. To achieve this, we design a top-down exploration strategy that analyzes historical workload $\mathcal{W}_{\text{hist}} = \{(Q_i, p_i, L_i(p_i))\}_{i=1}^{t}$ and extracts the most valuable common patterns (i.e., template queries) from them. Our algorithm comprises two main components: a historical workload analyzer and replacement rules.

Historical Workload Analysis. We recast Objective 1 as maximizing performance under a fixed overhead, with the optimization space bounded by a template budget $[\min_T, \max_T]$. The goal is to pinpoint the traditional optimizer's most suboptimal decisions, i.e., regions with the highest improvement potential, and prioritize them for learning. Rather than exhaustively probing all decisions, we adopt a top-down selection (lines 4–10) for two reasons: (i) improving ranking at the top layer most directly changes the final plan choice, and (ii) top-layer templates cover larger subspaces, yielding higher leverage per unit overhead.

Specifically, for an incoming set of historical workload $\mathcal{W}_{\text{hist}} = \{(Q_i, p_i, L_i(p_i))\}_{i=1}^{t}$, which consists of execution plans p and their corresponding queries Q, we can extract a series of sub-queries $q_1, q_2, \ldots, q_n$, each corresponding to a sub-plan $p_{q_1}, p_{q_2}, \ldots, p_{q_n}$ (line 5). These sub-plans are then organized based on their search level in DP, specifically sorted by the number of join relations they involve, from the largest to the smallest. For example, in Fig. 1, the root sub-plan joins $\{R_1, R_2, R_3\}$ at *Level 3*, whose children are level-2 sub-plans over $\{R_1, R_2\}$ and $\{R_2, R_3\}$, and each of them is composed from single-relation sub-plans at *Level 1*. Our top-down exploration always starts from the highest level (e.g., the full join over $\{R_1, R_2, R_3\}$) and then descends. In the figure, we highlight the Level 2 equivalent set $\{R_1, R_2\}$ as an example of a *key equivalent set* where ML intervention is applied. This top-layer sub-plan is templated as T (line 6). T and its associated sub-plan p are then added to a waiting queue for further processing. If $T \in \mathcal{A}$ (the current optimization space), we simply append the incoming plan p to T's experience pool for updating the replacement criterion. Otherwise, T is considered as a candidate template and may be admitted or rejected according to the budgeted replacement policy.

Replacement Rules. The optimization space is maintained via a dynamic replacement policy over template queries derived from historical workloads. This policy aims to maximize performance gains under a fixed template budget. Initially, when the optimization space contains fewer than $\min_T$ template queries, we expand the space by adding the top-layer template queries T from each query plan Q in the historical workload, without removing any existing template queries in the optimization space. This initial expansion phase ensures the optimization space quickly accumulates a foundational set of potentially valuable templates. Once the optimization space exceeds $\min_T$, a replacement mechanism is triggered to strictly maintain the size within the bounds of $[\min_T, \max_T]$ (lines 7-9). Specifically, for each template T, we rank by its *average latency* so that higher-latency templates receive higher priority (more room for improvement). Let $\mathcal{I}(T) = \{\, Q \in \mathcal{W}_{\text{hist}} \mid \text{tpl}(Q) = T \,\}$,

$$\overline{L}(T) = \frac{1}{|\mathcal{I}(T)|} \sum_{Q \in \mathcal{I}(T)} L\big(Q, p^O(Q)\big), \tag{4}$$

where $L(Q, p)$ denotes collected latency and $p^O(Q)$ is the expert (baseline) plan (or the observed production plan $p_{\text{obs}}(Q)$ when $p^O(Q)$ is unavailable). After LEON$^+$ has run for a while, we can switch to a benefit-driven policy that ranks templates by their realized *improvement over the baseline*. Let $\mathcal{I}_{\text{live}}(T)$ be the set of live queries matching template

T observed during deployment, and let $\hat{p}(Q)$ be LEON$^+$'s chosen plan. We compute

$$\overline{L}(T) = \frac{1}{|\mathcal{I}_{\text{live}}(T)|} \sum_{Q \in \mathcal{I}_{\text{live}}(T)} \Big[L\big(Q, p^O(Q)\big) - L\big(Q, \hat{p}(Q)\big) \Big], \tag{5}$$

During replacement, we admit templates with larger $\overline{L}(T)$ and evict those with the smallest $\overline{L}(T)$ to maintain the $|A|$ within $[\min_T, \max_T]$.

Discussion. The proposed replacement policy is history-driven: as defined in Objective 1, it maximizes overall latency reduction under a fixed budget using evidence from past executions. Our design and observed performance gains are not explicitly attributed to any single scenario (e.g., cold start on novel workloads or schema drift). The replacement rule is scenario-agnostic by construction. When a deployment emphasizes a particular setting, the rule can be specialized accordingly, for example, using predictive (future–workload–aware) prioritization, adopting tail-sensitive objectives (p95/p99 aware scoring), applying bandit-style selection [25], or introducing time-decayed weights to capture seasonality. These variants can plug into the same replacement rule interface. Admittedly, as with most template-driven methods, we make no guarantees for *ad-hoc* queries that match no template. In such cases, we simply fall back to the expert optimizer until sufficient evidence accrues.

5.2 Experience collection

The plan exploration inherently connects closely to the plan enumeration, since the mixed cost model M_θ^R is tasked to find the optimal plans among enumerated plans i.e., from the equivalent set S. We sample plans from $\mathcal{C}(S)$ (the set of rule–enumerated candidate physical plans) to collect additional training data, thereby mitigating selection bias from observing only deployed plans. Later, the collected training data will be executed to collect corresponding execution feedback denoted as $E = \{(q, Q, p, C(p), \omega, L(p))\}$. The plan exploration strategy specifies how to select the valuable training data from S into experience E (lines 12-21).

A unique challenge in query optimization is that it is time-consuming to get the execution feedback [25, 50]. LEON$^+$ aims to collect a limited amount of additional execution feedback with our exploration techniques. Here we have two important considerations as the exploration criteria: (1) Plans in the optimization space matter more. This is intuitively correct because the ML-aided optimizer only calibrates the estimated cost of plans in the optimization space. Experience excluded from the optimization space can hardly improve the performance of the ML-aided optimizer. (2) In addition, the ML-aided optimizer should correct its errors that lead to a sub-optimal query plan. Ideally, if we know how confident the M_θ^R is to its estimation, the low-confidence predictions are more likely to make mistakes.

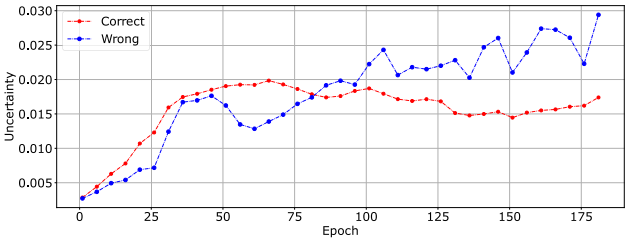

Fig. 4 An empirical study on the relationship between uncertainty and wrong samples. Wrong samples are defined by the contextual pair-wise classification problem

Model Uncertainty. Bayesian Neural Network (BNN) provides a framework to measure the uncertainty of a predictive model [17]. Different from the point estimation, given the prior distribution of model parameters $P(\theta)$, BNN learns the posterior distribution $p(\theta' \mid E)$ from experience E. When using a BNN for prediction, the probability distribution $p(\text{score} \mid LF, PF, E)$ can be marginalized by the posterior distribution $p(\theta' \mid E)$ for every θ:

$$p(\text{score} \mid LF, PF, E) = \int_\theta p(\text{score} \mid LF, PF, \theta') p(\theta' \mid E) d\theta'. \tag{6}$$

Based on Eq. (6), the model output can be approximated by $\mathbb{E}(\text{score} \mid LF, PF)$, which is calculated as follows.

$$\mathbb{E}(\text{score} \mid LF, PF) \approx \frac{1}{N} \sum_{i=1}^{N} M_\theta^R(LF, PF) \tag{7}$$

The uncertainty $u(LF, PF)$ for prediction (LF, PF) is defined as follows.

$$u(LF, PF) = Var(\text{score}) \approx \sum_{i=1}^{N} M_\theta^R(LF, PF)^2 - \mathbb{E}(\text{score} \mid LF, PF)^2 \tag{8}$$

N is the times of sampling the parameters from its posterior distribution. BNN is data-efficient as it can learn from limited data without overfitting [31].

We dig deep into the plan pairs and measure the uncertainty of the right and wrong classified plan pairs. We show an empirical study on the relationship between uncertainty and wrong samples. It can be shown from Fig. 4 that the uncertainty is unstable at the beginning of the training phase. With an increasing number of training epochs, the wrong

sample's uncertainty is obviously larger than the right sample, which indicates a positive relationship with estimation error.

Recent research in recommendation systems [9, 56] provides theoretical proof and empirical evidence that a high-ranked sample with larger uncertainty is helpful for model training. To this end, we propose a two-stage plan exploration strategy including top-down exploration and uncertainty-based exploration.

(Stage 1): Top-k Exploration. At the first stage, we choose the top $\lfloor k\% \times |S| \rfloor$ number of potential plans from the same equivalent set S, where $k\%$ is a tuneable parameter based on the training time budget.

(Stage 2): Uncertainty-based Exploration. In the second stage, for every plan we get from stage one, we select the most or several uncertain plans. We measure the uncertainty of the model to a plan denoted as $u(LF, PF)$ as a criterion to collect plans (line 14). When the ranking model learns from limited experience E, it will update the uncertainty to them and solicit for the more uncertain data samples. To measure $u(LF, PF)$ in a deep neural network, we adopt Monte Carlo dropout [17] by plugging dropout layers into M_θ^R. Based on Eq (6), we run M_θ^R for N times. We calculate its variance as an approximation to the uncertainty and calculate its mean as an approximation to its mixed cost estimation. The actual exploration number depends on the training time budget. Note that the exploration mode can be easily disabled by disabling dropout layers.

Tunable Exploration Budget. We expose a tunable exploration budget that operates at the equivalent-set granularity. During the resource idle windows, we evaluate up to k_{plan} high-uncertainty candidates per set under a global time cost budget k_{time} (line 11). This yields informative comparisons while capping exploration overhead and keeping service-level objectives intact.

5.3 Model training

In this section, we describe the training of the ranking model and the validation model. Through model training, the framework can better adapt to unseen workloads and consistently assist the query optimizer.

As described in Definition 6, we train the score function M_θ^R in a supervised classification manner under different contexts. To train model M_θ^R, we first pick several batches of plan pairs (p_1, p_2) satisfying $Q_1 = Q_1$ and $S_1 = S_2$, then we assign correct training labels. We assign the pair with a positive label if $L(p_1) < L(p_2)$. Otherwise, it will be assigned a negative label. The other model, the validation model, shares the same object but with different inputs (p, p^O) (i.e., the pair of a plan p and the optimal plan p^O chosen by the traditional query optimizer). Hence, the two models use the same loss function (i.e., cross-entropy loss), which we describe as follows.

Classification Loss. We adopt softmax binary cross entropy loss [6] as follows:

$$\begin{aligned} \mathcal{L}(p_1, p_2) &= -(y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})) \\ \hat{y} &= \sigma(\text{score}_{p_1} - \text{score}_{p_2}) \end{aligned} \tag{9}$$

where $\sigma(x) = \frac{1}{1+e^{-x}}$ is a *sigmoid* projection from the score to a probability to choose plan p_i, y is a true label by classifying a better plan in plan pair (p_1, p_2) based on the order of latency and $\hat{y}$ is the predicted label of the ML models. Then the cross entropy loss will penalize the wrong classification. Therefore, learning the loss $\mathcal{L}$ will make the calibrated cost model rank plan pair correctly.

Training the mixed cost model while maintaining its prior knowledge is not an easy task. Learning aggressively can result in unstable performance since the learned model tends to overfit on the limited data [53]. In the jargon of ML, researchers often include regularization to the objective function to avoid overfitting. In practice, we add KL divergence by measuring the difference in intra-equivalent set ranking before and after parameter updates as a soft constraint in our objective function similar to TRPO [39].

6 Integration

We implement a fine-grained ML-aided prototype on PostgreSQL and release it publicly. The design preserves plan-space completeness. We design two features as the extension of native PostgreSQL optimizer: per equivalent set activation and asynchronous bidirectional communication.

Per Equivalent Set Activation. We expose an equivalent-set level activation API inside the PostgreSQL optimizer that lets the ML side decide, *for each equivalent set S*, whether to engage ML filtering and ranking or fall back to the expert-only flow. Concretely, when S is populated, the optimizer emits a lightweight context record (template identifier, logical and physical properties) to the activation hook: ACTIVATE(S, ctx) $\rightarrow$ {ON, OFF}. If ON is set, the pipeline invokes M^V and M^R on S. Otherwise, the equivalent set proceeds unchanged. This mechanism realizes our *template-query* design at runtime: only S that match learned templates (i.e., belong to the optimization space) are switched on, enabling fine-grained control over overhead and risk (Section 4.3 and 5.1) while preserving plan-space completeness and the optimizer's DP invariants.

Asynchronous Bidirectional Communication. To avoid adding latency, LEON$^+$ decouples the ML side from the DBMS via an *asynchronous* bidirectional channel. LEON$^+$ implements a bidirectional communication channel between

Python and PostgreSQL, with a forward channel and a backward channel, respectively. On the forward path, the optimizer emits per–equivalent-set context and candidates plans for real-time serialization and concurrent exploration. On the backward path, the Python model returns filtered and ranked survivors that the optimizer can use immediately. Requests are issued as non-blocking calls and processed off the critical path. Plan serialization and model inference are overlapped with ongoing rule enumeration and expert costing, thereby largely hiding both costs from query-optimization latency. The same equivalent-set–level interface supports both *in-process* and *out-of-process* deployments: the former minimizes per-call latency; the latter provides isolation and hot-reload with minimal changes. Per–equivalent-set activation also applies in either mode. Our current implementation is based on inter-process communication (IPC). In the future, embedding the ranker in-process would eliminate IPC entirely, further reducing end-to-end optimization latency to be negligible relative to execution time.

LEON$^+$ is optimizer-agnostic and can integrate with any cost-based optimizer. Here we discuss how to integrate with Cascades-style optimizer.

Integration with Cascades Optimizers We integrate at the group (i.e., equivalence-set with required physical/logical property) level in a Cascades-style optimizer. After transformation rules populate a group G with a candidate set $\mathcal{C}(G)$, the optimizer first uses the conventional (expert) cost model to select $p^O(G) = \arg\min_{p\in\mathcal{C}(G)} c_{\exp}(p)$ purely as a reference baseline. Next, a lightweight validation model $\mathrm{M}^V(p,\ p^O(G))$ is applied to each candidate $p \in \mathcal{C}(G)$. Candidates predicted to perform worse than $p^O(G)$ are filtered out, yielding the survivor set $\mathcal{C}_{\text{valid}}(G) \subseteq \mathcal{C}(G)$. Then, an ML-based ranking module $\mathrm{M}^R(\cdot)$ takes $\mathcal{C}_{\text{valid}}(G)$ and produces a total order $\pi(G)$. The optimizer then picks $\hat{p}(G) = \text{top}\big(\pi(G)\big)$ as the plan for group G, and records it in the Memo structure. In this way the enumeration of logical and physical alternatives, enforcement of required physical properties (e.g., ordering, distribution), and the Memo architecture remain unchanged. LEON$^+$ augmentation only affects filtering of sub-optimal candidates and the selection order, with $p^O(G)$ serving as a baseline anchor rather than the final decision criterion.

7 Experiments

7.1 Experiment setup

Datasets. The four widely-used datasets listed below serve as benchmarks for the evaluation of improved LEON, (i.e., LEON$^+$):

- **Join Order Benchmark (JOB):** JOB is a real-world dataset that provides realistic workloads based on IMDB. There are 113 questions among 33 templates. It has 3.6GB of data (11GB when indexes are included) and 21 tables. The range of relations in each query is between 4 and 17.
- **Extended JOB (JOB-EXT)**: Ext-JOB is a demanding workload that presents a hard generalization challenge [26, 50]. The dataset consists of 24 new queries based on the IMDb dataset. Each query involves 2 to 10 joins, with an average of 5 joins per query. These queries are particularly challenging because they are out of distribution, meaning they utilize entirely different join templates and predicates compared to the original JOB.
- **STACK**: The STACK dataset is an extensive collection of over 18 million questions and answers sourced from 170 different StackExchange websites. The entire dataset occupies 100GB of storage space. For our purposes, we utilized the workload generated by [25], which includes 16 query templates. The number of relations in each query varies between 4 and 12.
- **TPC-H:** TPC-H is a database benchmark for industrial testing, including data obtained from decision support applications. It consists of eight tables and 61 columns and generates queries based on 22 templates. We produced 10GB of data.

We employed varying levels of difficulty in our training/test split to validate the effectiveness of LEON$^+$. For JOB and TPC-H benchmarks, we conducted evaluations under average circumstances. We randomly selected a query from each template as a test set, while the remaining queries were utilized for the training set. Furthermore, for the JOB-EXT benchmark, we examined the generalization of different methods to handle difficult, unseen queries. We used the JOB dataset for training and JOB-EXT for testing. Finally, we conducted tests on large-scale workloads for the STACK benchmark. We randomly select 100 queries for training and use 500 queries with distinct templates and predicates for testing.

Implementation and Environment. We use Python to implement ML models and algorithms. Our prototype keeps the ML side lightweight. The validation model M^V is a small MLP with 204K parameters ($\approx$ 0.82 MB in FP32), and the ranker M^R has 2.2M parameters ($\approx$ 8.95 MB in FP32). At

inference, the only additional state is the feature batch and two bounded async queues. By default, we use a 70-d per-plan summary, whose buffer is $O(B{\times}70{\times}w)$ bytes (batch size B, w=4 for FP32); e.g., with B=1000 and FP32 this is $\approx$0.27 MB. We hold a 666-d logical feature vector and 50-d per-node physical features; the peak buffer is $O\big(B{\cdot}(666+50\,\bar{n}){\cdot}w\big)$ for mean node count $\bar{n}$. For example, B=1000, $\bar{n}$=80, FP32 $\Rightarrow$ $\approx (666+50{\times}80){\times}4{\times}10^3 \approx 17.8MB$. Features are streamed and discarded after scoring, so the memory footprint is dominated by the $\sim$9 MB ranker loaded once per worker. All experiments are conducted on an Ubuntu server equipped with an Intel(R) Xeon(R) Silver 4214 2.20GHz CPU with 48 cores, 256GB DDR4 main memory, and a 1TB HDD.

Baselines. The baselines are shown below:

- **PostgreSQL.** PostgreSQL is an open-source DBMS, and we use it to represent the traditional method. PostgreSQL uses the histogram method to estimate the cost and then searches the execution plan by dynamic programming.
- **Balsa** [50]. Balsa is a query optimizer based on reinforcement learning and learned models, and we utilize its source code [36] to reproduce the results. For a fair comparison with LEON+, we use the expert cost model in the simulation stage, thus improving its performance. We configure Balsa in a non-parallel mode, with the same resource usage as other methods.
- **Bao** [25]. Bao uses ML models to aid the query optimizer of the DBMS in searching for an optimal execution plan. Similar to Bao's design, we obtain the final execution plan by letting Bao choose the hint set corresponding to the query statement, and record the result. In the same manner, as Balsa, we reproduce the results using the source code [37] of Bao.
- **LEON** [7]. our previous ML-aided query optimization framework with exploration based on ranking and uncertainty and ranking model-guided pruning.

Expert Engines. For a fair comparison, all learning-based query optimization methods are implemented based on PostgreSQL. Similar to previous works [20, 50], we set up PostgreSQL with 32GB shared buffers and cache size, along with 4GB work RAM, with GEQO turned off.

Evaluation Metrics. Unless otherwise specified, we report the workload runtime as an evaluation metric. The workload runtime is defined as the sum of latencies for each query. When presenting normalized runtimes, we calculate them with respect to the expert's runtimes. To demonstrate overall performance, we also report the Geometric Mean Relevant Latency (GMRL) which is adopted from [54], which is calculated as follows.

$$\text{GMRL} = \prod_{i=1}^{n} \frac{Latency(q)}{Latency_{expert}(q)} \tag{10}$$

The GMRL reflects the geometric average ratio of query execution time consumption between the learning-based model and the expert optimizer. A lower numerical value indicates better latency performance compared to the expert optimizer. Note that a GMRL value of 1 indicates expert optimizer latency performance. We chose to use GMRL because it can demonstrate the average performance without being affected by the latency of individual queries.

7.2 LEON+ performance

To demonstrate the overall performance of LEON+, we conduct end-to-end training on four benchmarks with different levels of difficulty in the training/test split. We ensure that all learning-based algorithms are trained until convergence, and we test them on the unseen test split. We repeat each end-to-end training 5 times and report average results.

Overall Latency Performance. Table 3 summarizes the overall latency performance of all baselines after training. In general, LEON+ achieves the best performance compared with PostgreSQL, Balsa, and Bao on JOB, JOB-EXT, STACK, and TPC-H. Note that LEON may have better performance than LEON+. This is because LEON+ focuses on the robustness and efficiency during optimization, instead of maximizing performance gains like LEON.

LEON+ outperforms PostgreSQL, with speedups of 1.63 $\times$, 1.23$\times$, 1.88$\times$, 1.55$\times$ in workload runtime for JOB, JOB-EXT, STACK, and TPC-H, respectively. TPC-H has the least improvement due to the evenly distributed data. These findings showcase the benefits of utilizing an ML-aided query optimizer, which improves the adaptability of expert optimizers to deployed datasets and workloads.

Compared with SOTA learning-based query optimizer, LEON+ also achieves the best query latency performance. Compared with Balsa, LEON+ achieves 1.19$\times$, 1.18$\times$, 1.30$\times$, 1.30$\times$ speedup in terms of workload runtime on JOB, JOB-EXT, STACK, and TPC-H respectively. Compared with Bao, LEON+ achieves 1.26$\times$, 1.03$\times$, 1.32$\times$, 1.48$\times$ speedup in terms of workload runtime on JOB, JOB-EXT, STACK, and TPC-H respectively. On JOB-EXT, Balsa shows the least improvement compared to ML-aided methods. That is because the ML-replaced methods lack essential knowledge in the expert query optimizer, which makes them less adaptable to changes in workload distribution. LEON+ achieves competitive performance compared to LEON. However, its runtime performance is slightly worse than LEON's on JOB-EXT and STACK. This is because LEON+ has a much more limited optimization space, which may sacrifice

Table 3 Overall performance of LEON$^+$ and baselines

Methods	Datasets							
	Workload Runtime (Seconds/Normalized)				GMRL			
	JOB	JOB-EXT	STACK	TPC-H	JOB	JOB-EXT	STACK	TPC-H
PostgreSQL	45.06/1.0	290.1/1.0	916.4/1.0	93.8/1.0	1.0	1.0	1.0	1.0
Balsa	32.81/0.72	279.1/0.96	637.2/0.69	78.7/0.76	0.62	1.01	0.67	0.74
Bao	34.68/0.77	243.5/0.83	646.5/0.70	89.2/0.95	0.95	1.08	1.13	0.94
LEON	28.54/0.63	**197.4/0.68**	**476.7/0.52**	66.3/0.70	**0.54**	**0.77**	**0.49**	0.72
LEON$^+$	**27.48/0.61**	234.9/0.81	486.7/0.53	**60.2/0.64**	0.56	0.85	0.73	**0.67**

performance in static scenarios but provides greater robustness.

The GMRL metric evaluates the overall relative latency performance of the queries regardless of their individual latencies. The results show that LEON$^+$ outperforms Balsa and Bao on two difficult benchmarks, JOB-EXT and STACK. Interestingly, Balsa and Bao show similar results to PostgreSQL in terms of GMRL, despite having lower workload runtimes. We observed that both of them tend to focus only on improving the slowest queries and neglecting other queries. In contrast, LEON$^+$ can improve the performance of the slowest queries while maintaining latency performance for other queries that have less optimization potential. This finding is also supported by the results presented in Section 7.2.

Query Regression Analysis. Figure. 5 presents the normalized runtime of each learning-based query optimizer (Balsa, Bao, LEON, and LEON$^+$) over PostgreSQL plans on a per-query basis. The x-axis denotes PostgreSQL expert runtime for every query. This figure provides an overview of the performance of each optimizer on individual queries. Additionally, for the STACK benchmark, we plot 100 test queries' performance from the test set to improve the visualization, while the overall trend remains the same.

Overall, Balsa, Bao, LEON, and LEON$^+$ reduce the latency of slow queries from PostgreSQL, which explains why they can outperform the expert query optimizer. It is worth mentioning that LEON and LEON$^+$ exhibit a significant reduction in query performance regression individually while maintaining similar performance on queries that are inherently fast to execute. Specifically, for the four benchmarks, Balsa causes 40% queries with performance regression. Bao causes 38% queries with performance regression. By contrast, LEON and LEON$^+$ only cause 19% and 17% queries with performance regression, respectively. In terms of the extent of the performance regression, Balsa and Bao have 7.4× and 9.6× slowdown in query performance on JOB and STACK respectively, while LEON$^+$ only causes up to 1.6× slowdown among four benchmarks.

LEON reduces such performance regression for two reasons: 1) Compared to ML-replaced methods, LEON maintains the expert query optimizer knowledge as much as possible. 2) Compared to ML-aided methods, LEON learns to rank effectively instead of predicting the absolute latency, which introduces less error to the prediction. In addition, LEON, as a white-box method, deconstructs the search space and concentrates on the crucial aspect of query optimization (such as higher-ranked plans), which also makes the ranking model learn faster and generate effective predictions.

LEON$^+$ further reduces performance regression from two aspects. 1) Even though ML methods often lack interpretability, LEON$^+$ can still effectively manage the optimization space of ML components within a traditional optimizer. Such approach brings enhanced robustness. 2) LEON$^+$'s lightweight validation model reduces performance regression by filtering out suboptimal execution plans.

Training Efficiency. In this section, we analyze the training efficiency of LEON$^+$, i.e., the change of the test query performance with the training time. Fig. 6 shows the training curve with the variance of learning-based query optimizers on JOB, JOB-EXT, STACK, and TPC-H. The shaded area represents the range between the minimum and maximum values obtained from five different runs using different random seeds.

LEON$^+$ achieves efficient training on four benchmarks. LEON$^+$ outperforms PostgreSQL consistently by about 2.5 hours, 3 hours, 3 hours, and 1 hour on JOB, JOB-EXT, STACK, and TPC-H, respectively. Compared to Balsa, LEON$^+$ demonstrates superior initial performance, lower variance, and faster convergence during training. This highlights the inherent superiority of ML-aided methods over ML-replaced methods, as the former can leverage more basic knowledge from the expert optimizer.

As training time increases, the performance gap between LEON$^+$ and Bao widens, particularly on two challenging workloads: JOB-EXT and STACK. On JOB and TPC-H, Bao outperforms LEON$^+$ at the beginning. However, after several hours, LEON$^+$ outperforms Bao consistently. This demonstrates that the upper limit of white-box methods is higher and LEON$^+$ effectively explores potential better query plans.

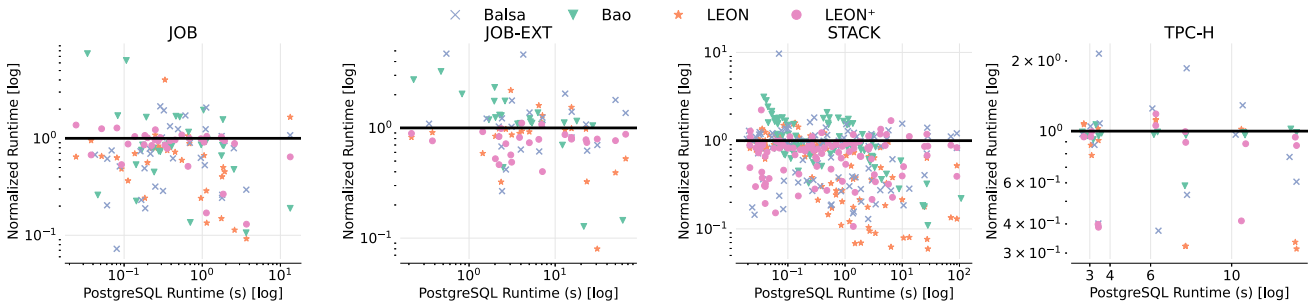

Fig. 5 Breakdown of LEON+'s per-query performance compared to the PostgreSQL runtime on different datasets

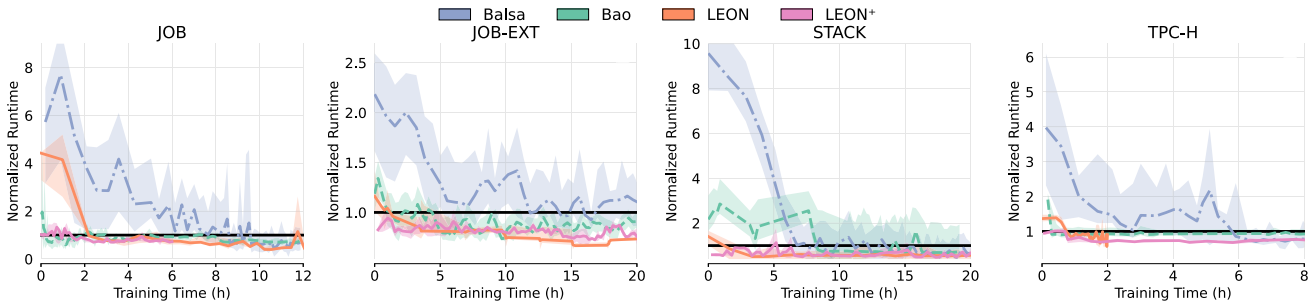

Fig. 6 Training curves with variance on different datasets. The shaded area represents the range between the minimum and maximum values obtained from five different runs using different random seeds

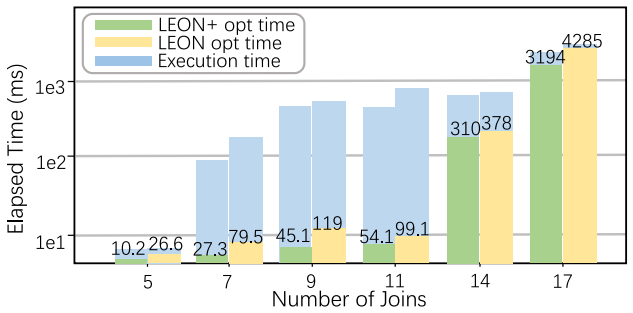

Fig. 7 Breakdown of optimization and execution time on a different number of join tables

The performance of LEON+ remains robust compared to other learning-based methods during the training process. In comparison to Bao and Balsa, LEON+ exhibits more consistent performance and lower variability. This is due to our contextual learning-to-rank objective, which is inherently suited to the query optimization problem. Additionally, this objective assists in the validation and debugging of our ML models. Compared to LEON, LEON+ effectively eliminates cold start issues even further. From the early stages of training, LEON+ consistently outperforms LEON, due to its robust control over ML modules.

7.3 Optimization time

We evaluate optimizer (opt) time and execution time on JOB benchmark, stratified by join count, and report the results in Fig. 7 (log scale; blue: execution time, yellow: LEON opt time, green: LEON+ opt time). For small–medium queries (5–11 joins), LEON+ adds only tens of milliseconds of planning overhead (10.2, 27.3, 45.1, 54.1 ms, respectively), which is two orders of magnitude below execution time. For large joins, where planning becomes non-negligible, LEON+ substantially reduces optimizer time relative to LEON, e.g., 310 ms vs. 378 ms at 14 joins (−18%) and 3194 ms vs. 4285 ms at 17 joins (−25%). Overall, LEON+ preserves the execution-time benefits of ML-guided planning while cutting optimization overhead compared to LEON, especially in the high-join regime.

7.4 Adapting to dynamic workload

In this section, we demonstrate how LEON+ adapts to a dynamic workload. To illustrate changes in the workload, we use a "time series split" strategy [25]. Specifically, we introduce queries to the optimizer incrementally, one at a time. The learned optimizers are consistently evaluated on queries that they have not encountered during model updating. Unlike previous work, we have limited queries and begin measuring performance from scratch. We conduct dynamic workload on four datasets: JOB, JOB_TRAINING, TPC-H, and STACK.

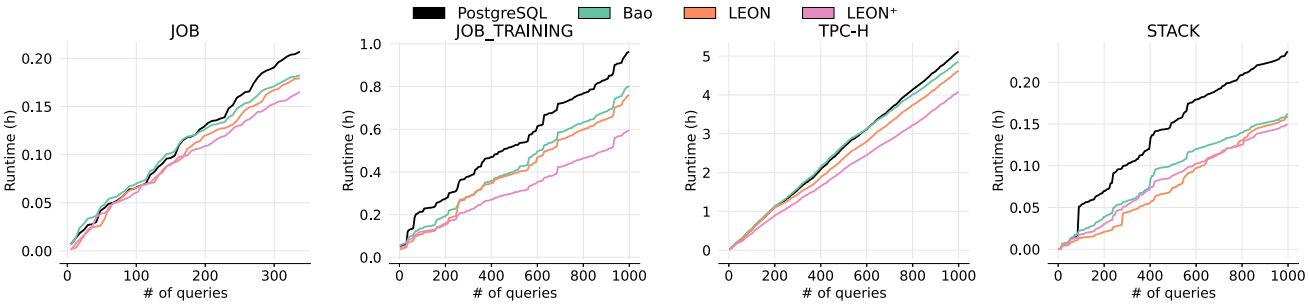

Fig. 8 Runtime with dynamic unseen workload on different datasets

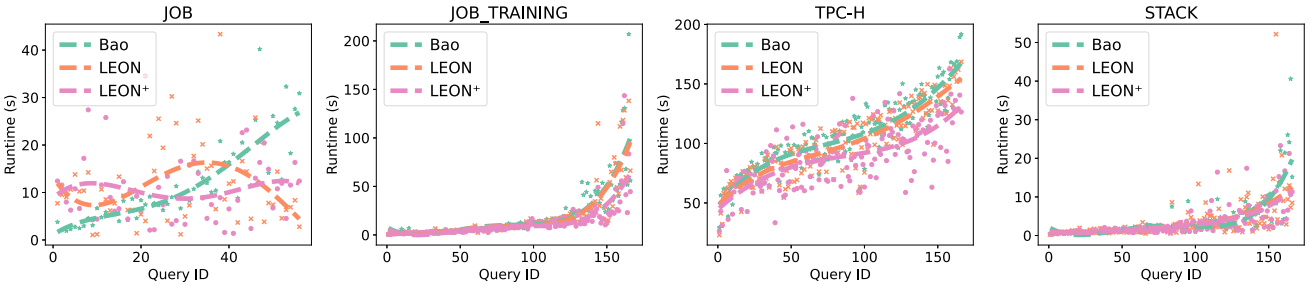

Fig. 9 Absolute differences in a subset of dynamic unseen query runtimes across various learning-based methods. The x-axis is sorted based on the default PostgreSQL runtime for each query point. The dashed line represents the trend in execution time variation for each method

Since the JOB dataset only contains 113 queries, we traverse these queries a few times to provide a workload with more training queries. JOB_TRAINING consists of 1000 queries derived from the original JOB workload. We only report Bao, LEON, and PostgreSQL as baselines because Balsa does not achieve competitive performance.

Overall, LEON$^+$ consistently outperforms PostgreSQL and Bao during training, with substantially less initial performance regression. The runtime gap between LEON$^+$ and the baselines keeps growing for two reasons. First, utilizing the shared knowledge of queries, the robust generalization in LEON$^+$ contributes to being effective to new queries. Second, since the template queries are adaptively managed during the dynamic workload, LEON$^+$ can potentially focus on the important optimization space for the recent queries. Another phenomenon shown in Fig. 8 is that, although LEON has better performance in runtime at the beginning, LEON$^+$ can surpass LEON with more queries. This is because the top-down exploration strategy in LEON$^+$ retains the optimal plans, while the exploration strategy in LEON may generate unstable query plans during a dynamic workload.

Robustness Analysis. Fig. 9 shows the absolute differences in query runtimes across various learning-based methods during dynamic scenarios. Points further to the right represent tail latency, which is a concern to users. From both the trend line and point observations, it is evident that LEON$^+$ not only consistently outperforms baseline methods but also significantly reduces tail latency in dynamic scenarios. On JOB, which has relatively short queries, LEON$^+$ achieves stable performance to improve tail-latency queries. On STACK, LEON$^+$ achieves up to 2.24× tail latency reduction compared to LEON. This is because LEON$^+$ has a better ML component management and aims to reduce performance regression. This improvement underscores LEON$^+$'s effectiveness in managing high-latency queries, providing a more responsive and reliable performance under changing workloads.

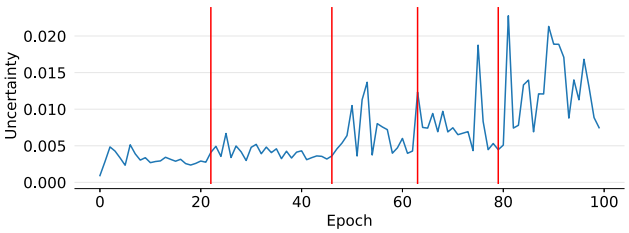

Fig. 10 Uncertainty measurement for dynamic workload

Effectiveness of Uncertainty. Fig. 10 shows the fluctuations in uncertainty estimated by ML models during training. The red line demarcates different stages. In each stage, the uncertainty initially increases due to unseen data and then decreases as more training iterations are performed. This finding highlights the adaptability of LEON$^+$ in scenarios where system dynamics are changing over time. In such cases, LEON$^+$'s uncertainty estimates can enable the expert optimizer to identify system changes and modify its decisions about updating ML models accordingly.

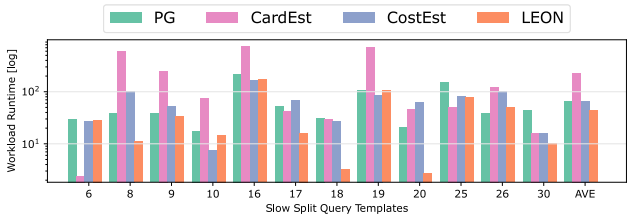

Fig. 11 Workload Runtime of different templates on Slow-split JOB

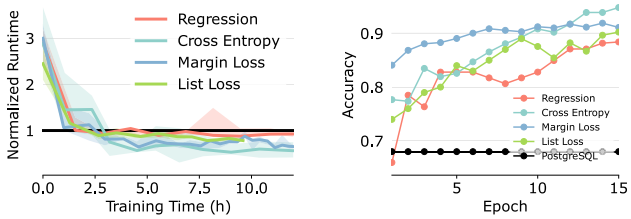

Fig. 12 Impact of different ranking models

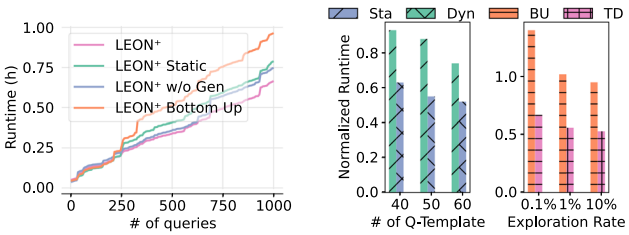

Fig. 13 Effect of query templates exploration and exploration rate

7.5 Comparison with white-box methods

Here, LEON+ compares its performance with white-box methods, including CardEst and CostEst. We have implemented open-source SOTA learning-based methods, such as NeuroCard [52] as the CardEst work, and Tpool [43] as the CostEst. NeuroCard uses the IMDB dataset as training data, while we have used training queries provided by [43] for Tpool. We have injected their predicted cardinality and cost into PostgreSQL (PG) DP search to measure end-to-end latency performance. Similar to [50], we have split the slowest queries from JOB as the test workload. Fig. 11 shows the detailed workload runtime of different methods on every query template, and the last bar shows the average result. On average, LEON+ outperforms PG, CostEst, and CardEst by about 33%, 34%, and 81%, respectively. LEON+ outperforms the baselines due to its accurate ranking-based model and exploration strategy. CardEst does not outperform CostEst for two reasons. 1) The training dataset for Tpool is large enough to boost the performance of ML models. 2) CardEst methods can be influenced by PG's inaccurate cost model, which has been studied in [20].

7.6 Ablation study

Different Ranking Models. In this section, we will analyze the ranking objective of LEON+, which includes three types: pointwise, pairwise, and listwise. We have implemented Regression as a pointwise method, MarginLoss [23] and Cross Entropy as two pairwise methods, and List Loss [46] as a listwise method. Fig. 12 (left) displays the training curve of different ranking models. We can observe that the Regression method has inferior performance and more fluctuation, which is expected as proven by other works in recommendation systems. List Loss outperforms Regression, and the pairwise methods have the best performance. This is because it is unrealistic to collect all plans' execution feedback in an equivalent set, while pairwise learning avoids such shortcomings. This result shows pairwise ranking is more suitable for query optimization. Fig. 12 (right) shows the accuracy of contextual pairwise ranking ML models trained by different ranking models. The accuracy results of different methods also demonstrate our conclusion.

Effect of Exploration Strategy. We further evaluate the effectiveness of the adaptive query templates, the generalization with shared knowledge, and the top-down exploration strategy. The ablation study is conducted by comparing LEON+ model against three alternative models: 1) LEON+ Static, which utilizes fixed query templates to control a fixed optimization space. 2) LEON+ w/o Gen, which directly keeps extensive initially generated plans without filtering for pairwise training. 3) LEON+ Bottom Up, which uses a bottom-up exploration strategy in the plan space. We record their performance in runtime in Fig. 13 (left). We can see that LEON+ model outperforms the LEON+ Static, LEON+ w/o Gen, and LEON+ Bottom-up. This provides strong evidence that the adaptive query templates, the generalization training design, and the top-down exploration strategy enhance the performance of the ML-aided optimizer.

Effect of Query Templates and Exploration Rate. As for the choice of parameters, we analyze the setting of the number of query templates and exploration rate. As shown in Fig. 13(right), we compare the performance of query templates maintained statically (Sta) and dynamically (Dyn) respectively, and the performance with exploration in the bottom-up (BU) and top-down (TD) way individually. Query templates maintained in Dyn outperform those in Sta continuously, because the dynamic query templates are more likely to shift along with the change of queries. Exploration in TD also achieves less runtime than in BU among all exploration rates, since it ensures the generation of optimal plans. Due to the law of diminishing marginal, the additional generated plans tend to be timeout with higher parameters than 60 query templates and 10% exploration rate. Hence, the number of query templates is maintained within (55, 65), and the exploration rate is set as 10% in this work.

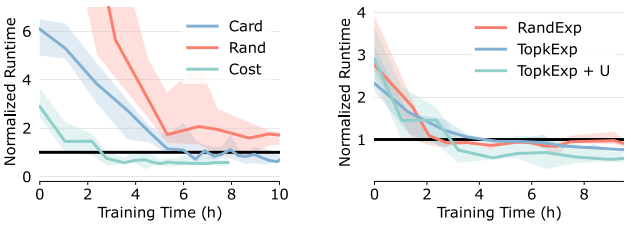

Fig. 14 Impact of prior knowledge and impact of exploration strategies

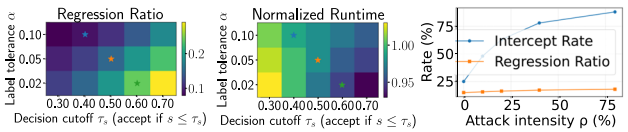

Fig. 15 The effectiveness of the validation model

Different Prior Knowledge. In this context, "prior knowledge" refers to using information that is available before the model is trained, such as empty knowledge basis (Rand), the estimated cardinality (Card), and the expert cost model (Cost). In LEON$^+$, we use these three initializations for ML models. In the case of Fig. 14 (left), the effect of choosing different priors is analyzed by comparing the convergence speed and latency of the ML-aided optimizer on the JOB benchmark when using different types of prior knowledge. The results show that using prior knowledge can improve the convergence speed of the model, and using cost-based correction can prevent the generation of bad plans (performance regression). Therefore, incorporating prior knowledge into the model can improve its performance. This helps to greatly alleviate the cold-start problem, which can be a significant challenge for ML models deployed in real-world systems.

Different Sampling Strategies. In Fig. 14 (right), three different sampling strategies are compared based on the training process on JOB: "RandExp", which randomly samples plans for training, "TopkExp", which selects the top $k\%$ of plans based on LEON$^+$'s score, and "TopkExp+U", which selects plans with larger variance in addition to the top $k\%$ of plans based on LEON$^+$'s score. The results show that "TopkExp+U" has the fastest convergence speed and the lowest latency. This is because the uncertainty measure is used to further screen out data samples with little training value, which helps to avoid wasting training time on worthless samples. In this way, the model can focus on learning the core knowledge of the data, which improves its generalization ability.

Effect of Validation Model. We ablate the validation gate by sweeping the decision cutoff $\tau_s \in \{0.30, 0.40, 0.50, 0.60, 0.70\}$ and the label tolerance $\alpha \in \{0.10, 0.05, 0.02\}$, reporting two heatmaps in Fig. 15: *Regression Ratio* (left; lower is safer) and *Normalized Runtime* (middle; lower is faster, measured relative to our default setting). As expected, for a fixed α, increasing τ_s admits more candidates, which raises regressions but reduces runtime; for a fixed τ_s, using a larger tolerance (softer labeling) lowers regressions and modestly improves runtime. We highlight three operating points: **conservative** ($\tau_s = 0.40, \alpha = 0.10$) with 9% regressions, **balanced** (0.50, 0.05) with 15%, and **aggressive** (0.60, 0.02) with 24%. This plot makes the trade-off explicit and motivates using the balanced point by default unless a deployment prefers stronger safety (conservative) or higher potential gains (aggressive).

To study the effect of the validation model further, we conduct adversarial reordering of candidate plans shown in Fig. 15 (right). We stress-test the validation model gate by targeting *ranker-induced misorderings*. For each equivalent set S, we identify candidates that the ranking model erroneously promotes (i.e., labeled as deteriorating with $\Delta_{\text{rel}} > \alpha$ but placed ahead by the ranker), and *inject* a $\rho\%$ subset of these candidates at the head of $\mathcal{C}(S)$ to form an adversarial front. All results are reported at the balanced operating point (decision cutoff τ_s=0.50, label tolerance α=0.05). The figure plots two metrics against $\rho \in \{0, 10, 20, 40, 80\}$: (i) the Intercept Rate–the fraction of promoted bad candidates filtered by M^V–and (ii) the Regression Ratio–the fraction of final plans with $\Delta_{\text{rel}}(p, p^O) > \alpha$. As attack intensity increases, M^V blocks an ever larger portion of harmful reorderings (intercept 25% → 48% → 63% → 78% → 88%), while the overall regression rises only mildly (15.0% → 15.6% → 16.3% → 17.2% → 18.0%). This indicates that the validation gate effectively safeguards against adversarial reorderings: most promoted deteriorating candidates are intercepted, and the residual risk grows slowly with ρ.

8 Related work

ML-aided Query Optimizer. Leo [29] is the pioneer work that makes use of learning-based concepts to assist the query optimizer and advance it. Leo proposes to collect more statistics for the optimizer histogram during the query execution. Encouraged by the recent popularity of ML, many researchers apply ML techniques to help resolve subproblems in the query optimizer. Data-driven methods like [45, 49, 51, 59] and query-driven methods like [10, 33] are proposed to solve the cardinality estimation. Deep neural networks [42, 43] are trained in a supervised learning fashion to resolve cost estimation. Reinforcement learning (RL) helps solve decision-making problems such as database tuning problems [22, 44, 55]. Bao and its variants [25, 32, 57] propose to tune hint sets for each query, which is promising for practical usage. Recent ML-based optimizers, such as Behr et al. [4], reformulate cost estimation as a learning-to-rank task to replace the cost model, whereas LEON$^+$ focuses on

maximizing the retention of expert knowledge while actively exploring unseen experiences.

Learned Query Optimizer. Recently, RL has been applied to learn an optimizer to generate query execution plans. Neo [26] builds an end-to-end query optimizer that produces complete execution plans. However, Neo is trained completely based on latency signals, which requires DBMS to execute numerous plans including potentially bad ones. Some other similar works including Rejoin [27], DQ [18], and RTOS [54] leverage cost as a trade-off to increase training efficiency and then transfer the pre-trained model based on the cost to a new model that can adapt to latency signals. DQ and RTOS leverage inductive transfer learning methods [35] that change representations in the output layer. Balsa [50] shows the insight of learning an optimizer without the expert and achieves SOTA performance.

9 Conclusion

In this paper, we propose LEON+, a framework for ML-aided expert optimization. Different from the existing learning-based methods, LEON+ trains a ranking model based on the fundamental knowledge of the expert query optimizer and aims to help the expert query optimizer self-adjust to the deployment environment. By adapting the optimization space, we significantly enhance LEON+'s robustness against unseen workloads and reduce exploration costs. We conducted extensive experiments on four public benchmarks, providing evidence that LEON+ exhibits superior performance in terms of execution latency, training efficiency, and stability. We also integrate the LEON+ framework into traditional optimizers with an unobtrusive and automated integration.

Acknowledgements This work is partially supported by NSFC (No. 62272086, No. 62472068) and Municipal Government of Quzhou under Grant (No. 2023D004, 2023Z003, 2023D043).

References

1. Akdere, M., Çetintemel, U., Riondato, M., Upfal, E., Zdonik, S.B.: Learning-based query performance modeling and prediction. In: 2012 IEEE 28th International Conference on Data Engineering, pp. 390–401. IEEE (2012)
2. Basu, D., Lin, Q., Chen, W., Vo, H.T., Yuan, Z., Senellart, P., Bressan, S.: Cost-model oblivious database tuning with reinforcement learning. In: Database and Expert Systems Applications (2015)
3. Basu, D., Lin, Q., Chen, W., Vo, H.T., Yuan, Z., Senellart, P., Bressan, S.: Regularized cost-model oblivious database tuning with reinforcement learning. In: Transactions on Large-Scale Data- and Knowledge-Centered Systems XXVIII, pp. 96–132. Springer (2016)
4. Behr, H., Markl, V., Kaoudi, Z.: Learn what really matters: A learning-to-rank approach for ml-based query optimization. In: BTW 2023, pp. 535–554. Gesellschaft für Informatik eV (2023)
5. Bharadhwaj, H.: Meta-learning for user cold-start recommendation. In: 2019 International Joint Conference on Neural Networks (IJCNN), pp. 1–8. IEEE (2019)
6. Cao, Z., Qin, T., Liu, T.Y., Tsai, M.F., Li, H.: Learning to rank: from pairwise approach to listwise approach. In: Proceedings of the 24th international conference on Machine learning, pp. 129–136 (2007)
7. Chen, X., Chen, H., Liang, Z., Liu, S., Wang, J., Zeng, K., Su, H., Zheng, K.: Leon: a new framework for ml-aided query optimization. Proceedings of the VLDB Endowment **16**(9), 2261–2273 (2023)
8. Ding, B., Das, S., Marcus, R., Wu, W., Chaudhuri, S., Narasayya, V.R.: Ai meets ai: Leveraging query executions to improve index recommendations. In: Proceedings of the 2019 International Conference on Management of Data, pp. 1241–1258 (2019)
9. Ding, J., Quan, Y., Yao, Q., Li, Y., Jin, D.: Simplify and robustify negative sampling for implicit collaborative filtering. Adv. Neural. Inf. Process. Syst. **33**, 1094–1105 (2020)
10. Dutt, A., Wang, C., Nazi, A., Kandula, S., Narasayya, V., Chaudhuri, S.: Selectivity estimation for range predicates using lightweight models. Proc. VLDB Endow. **12**(9), 1044–1057 (2019)
11. Ganaie, M.A., Hu, M., Malik, A.K., Tanveer, M., Suganthan, P.N.: Ensemble deep learning: A review. Eng. Appl. Artif. Intell. **115**, 105151 (2022)
12. Goodfellow, I.J., Mirza, M., Xiao, D., Courville, A., Bengio, Y.: An empirical investigation of catastrophic forgetting in gradient-based neural networks (2013). arXiv preprint arXiv:1312.6211
13. Graefe, G.: The cascades framework for query optimization. IEEE Data Eng. Bull. **18**(3), 19–29 (1995)
14. Graefe, G., McKenna, W.J.: The volcano optimizer generator: Extensibility and efficient search. In: Proceedings of IEEE 9th international conference on data engineering, pp. 209–218. IEEE (1993)
15. Han, Y., Wu, Z., Wu, P., Zhu, R., Yang, J., Tan, L.W., Zeng, K., Cong, G., Qin, Y., Pfadler, A., et al.: Cardinality estimation in dbms: A comprehensive benchmark evaluation (2021). arXiv preprint arXiv:2109.05877
16. Jindal, A., Qiao, S., Sen, R., Patel, H.: Microlearner: A fine-grained learning optimizer for big data workloads at microsoft. In: 2021 IEEE 37th International Conference on Data Engineering (ICDE), pp. 2423–2434. IEEE (2021)
17. Jospin, L.V., Laga, H., Boussaid, F., Buntine, W., Bennamoun, M.: Hands-on bayesian neural networks-a tutorial for deep learning users. IEEE Comput. Intell. Mag. **17**(2), 29–48 (2022)
18. Krishnan, S., Yang, Z., Goldberg, K., Hellerstein, J., Stoica, I.: Learning to optimize join queries with deep reinforcement learning (2018). arXiv preprint arXiv:1808.03196
19. Lee, K., Dutt, A., Narasayya, V., Chaudhuri, S.: Analyzing the impact of cardinality estimation on execution plans in microsoft sql server. Proceedings of the VLDB Endowment **16**, 2871–2883 (2023)
20. Leis, V., Gubichev, A., Mirchev, A., Boncz, P., Kemper, A., Neumann, T.: How good are query optimizers, really? Proceedings of the VLDB Endowment **9**(3), 204–215 (2015)
21. Li, G., Zhou, X., Cao, L.: Ai meets database: Ai4db and db4ai. In: Proceedings of the 2021 International Conference on Management of Data, pp. 2859–2866 (2021)
22. Li, G., Zhou, X., Li, S., Gao, B.: Qtune: A query-aware database tuning system with deep reinforcement learning. Proc. VLDB Endow. **12**(12), 2118–2130 (2019)
23. Liu, L., Dou, Q., Chen, H., Qin, J., Heng, P.A.: Multi-task deep model with margin ranking loss for lung nodule analysis. IEEE Trans. Med. Imaging **39**(3), 718–728 (2019)

24. Ma, L., Ding, B., Das, S., Swaminathan, A.: Active learning for ml enhanced database systems. In: Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (2020)
25. Marcus, R., Negi, P., Mao, H., Tatbul, N., Alizadeh, M., Kraska, T.: Bao: Making learned query optimization practical. ACM SIGMOD Rec. **51**(1), 6–13 (2022)
26. Marcus, R., Negi, P., Mao, H., Zhang, C., Alizadeh, M., Kraska, T., Papaemmanouil, O., Tatbul, N.: Neo: A learned query optimizer (2019). arXiv preprint arXiv:1904.03711
27. Marcus, R., Papaemmanouil, O.: Deep reinforcement learning for join order enumeration. In: Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management, pp. 1–4 (2018)
28. Marcus, R., Papaemmanouil, O.: Plan-structured deep neural network models for query performance prediction (2019). arXiv:1902.00132
29. Markl, V., Lohman, G.M., Raman, V.: Leo: An autonomic query optimizer for db2. IBM Systems Journal (2003)
30. Moerkotte, G., Neumann, T., Steidl, G.: Preventing bad plans by bounding the impact of cardinality estimation errors. Proc. VLDB Endow. **2**(1), 982–993 (2009)
31. Mullachery, V., Khera, A., Husain, A.: Bayesian neural networks (2018). arXiv preprint arXiv:1801.07710
32. Negi, P., Interlandi, M., Marcus, R., Alizadeh, M., Kraska, T., Friedman, M., Jindal, A.: Steering query optimizers: A practical take on big data workloads. In: Proceedings of the 2021 International Conference on Management of Data (2021)
33. Negi, P., Marcus, R., Kipf, A., Mao, H., Tatbul, N., Kraska, T., Alizadeh, M.: Flow-loss: learning cardinality estimates that matter (2021). arXiv preprint arXiv:2101.04964
34. Ortiz, J., Balazinska, M., Gehrke, J., Keerthi, S.S.: An empirical analysis of deep learning for cardinality estimation (2019). arXiv preprint arXiv:1905.06425
35. Pan, S.J., Yang, Q.: A survey on transfer learning. IEEE Transactions on knowledge and data engineering (2009)
36. balsa project: https://github.com/balsa-project/balsa (2022)
37. RyanMarcus: https://github.com/learnedsystems/BaoForPostgreSQL
38. Saxena, G., Rahman, M., Chainani, N., Lin, C., Caragea, G., Chowdhury, F., Marcus, R., Kraska, T., Pandis, I., Narayanaswamy, B.: Auto-wlm: Machine learning enhanced workload management in amazon redshift. In: Companion of the 2023 International Conference on Management of Data, pp. 225–237 (2023)
39. Schulman, J., Levine, S., Abbeel, P., Jordan, M., Moritz, P.: Trust region policy optimization. In: International conference on machine learning, pp. 1889–1897. PMLR (2015)
40. Selinger, P.G., Astrahan, M.M., Chamberlin, D.D., Lorie, R.A., Price, T.G.: Access path selection in a relational database management system. In: Proceedings of the 1979 ACM SIGMOD international conference on Management of data, pp. 23–34 (1979)
41. Sharma, A., Schuhknecht, F.M., Dittrich, J.: The case for automatic database administration using deep reinforcement learning (2018). arXiv preprint arXiv:1801.05643
42. Siddiqui, T., Jindal, A., Qiao, S., Patel, H., Le, W.: Cost models for big data query processing: Learning, retrofitting, and our findings. In: Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (2020)
43. Sun, J., Li, G.: An end-to-end learning-based cost estimator (2019). arXiv preprint arXiv:1906.02560
44. Van Aken, D., Yang, D., Brillard, S., Fiorino, A., Zhang, B., Bilien, C., Pavlo, A.: An inquiry into machine learning-based automatic configuration tuning services on real-world database management systems. Proceedings of the VLDB Endowment (2021)
45. Wang, J., Chai, C., Liu, J., Li, G.: Face: a normalizing flow based cardinality estimator. Proceedings of the VLDB Endowment (2021)
46. Wang, X., Hua, Y., Kodirov, E., Hu, G., Garnier, R., Robertson, N.M.: Ranked list loss for deep metric learning. In: Proceedings of the IEEE/CVF conference on computer vision and pattern recognition, pp. 5207–5216 (2019)
47. Wu, C., Jindal, A., Amizadeh, S., Patel, H., Le, W., Qiao, S., Rao, S.: Towards a learning optimizer for shared clouds. Proceedings of the VLDB Endowment **12**(3), 210–222 (2018)
48. Wu, Z., Marcus, R., Liu, Z., Negi, P., Nathan, V., Pfeil, P., Saxena, G., Rahman, M., Narayanaswamy, B., Kraska, T.: Stage: Query execution time prediction in amazon redshift. In: Companion of the 2024 International Conference on Management of Data, pp. 280–294 (2024)
49. Wu, Z., Shaikhha, A.: Bayescard: A unified bayesian framework for cardinality estimation. arXiv e-prints (2020)
50. Yang, Z., Chiang, W.L., Luan, S., Mittal, G., Luo, M., Stoica, I.: Balsa: Learning a query optimizer without expert demonstrations (2022). arXiv preprint arXiv:2201.01441
51. Yang, Z., Kamsetty, A., Luan, S., Liang, E., Duan, Y., Chen, X., Stoica, I.: Neurocard: one cardinality estimator for all tables (2020). arXiv preprint arXiv:2006.08109
52. Yang, Z., Kamsetty, A., Luan, S., Liang, E., Duan, Y., Chen, X., Stoica, I.: NeuroCard: One cardinality estimator for all tables. pp. 61–73. VLDB Endowment (2021)
53. Ying, X.: An overview of overfitting and its solutions. In: Journal of physics: Conference series, vol. 1168, p. 022022. IOP Publishing (2019)
54. Yu, X., Li, G., Chai, C., Tang, N.: Reinforcement learning with tree-lstm for join order selection. In: 2020 IEEE 36th International Conference on Data Engineering (ICDE) (2020)
55. Zhang, J., Liu, Y., Zhou, K., Li, G., Xiao, Z., Cheng, B., Xing, J., Wang, Y., Cheng, T., Liu, L., et al.: An end-to-end automatic cloud database tuning system using deep reinforcement learning. In: Proceedings of the 2019 International Conference on Management of Data (2019)
56. Zhang, W., Chen, T., Wang, J., Yu, Y.: Optimizing top-n collaborative filtering via dynamic negative item sampling. In: Proceedings of the 36th international ACM SIGIR conference on Research and development in information retrieval, pp. 785–788 (2013)
57. Zhang, W., Interlandi, M., Mineiro, P., Qiao, S., Ghazanfari, N., Lie, K., Friedman, M., Hosn, R., Patel, H., Jindal, A.: Deploying a steered query optimizer in production at microsoft. In: Proceedings of the 2022 International Conference on Management of Data (2022)
58. Zhou, X., Chai, C., Li, G., Sun, J.: Database meets artificial intelligence: A survey. IEEE Transactions on Knowledge and Data Engineering (2020)
59. Zhu, R., Wu, Z., Han, Y., Zeng, K., Pfadler, A., Qian, Z., Zhou, J., Cui, B.: Flat: fast, lightweight and accurate method for cardinality estimation (2020). arXiv preprint arXiv:2011.09022