LEON: A New Framework for ML-Aided Query Optimization

Xu Chen*
University of Electronic Science and Technology of China
xuchen@std.uestc.edu.cn

Haitian Chen
University of Electronic Science and Technology of China
haitianchen@std.uestc.edu.cn

Zibo Liang
University of Electronic Science and Technology of China
zbliang@std.uestc.edu.cn

Shuncheng Liu
University of Electronic Science and Technology of China
liushuncheng@std.uestc.edu.cn

Jinghong Wang
Huawei Technologies Co., Ltd.
jh.wang@huawei.com

Kai Zeng
Huawei Technologies Co., Ltd.
kai.zeng@huawei.com

Han Su†
University of Electronic Science and Technology of China
hansu@uestc.edu.cn

Kai Zheng†
University of Electronic Science and Technology of China
zhengkai@uestc.edu.cn

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-aid*E*d query Optimizatio*N*. *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. To train the ML model, a pairwise ranking objective is proposed, which is substantially different from the previous regression objective. To help the optimizer to 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. Furthermore, an ML model-guided pruning is proposed to increase the planning efficiency without hurting too much performance. 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.

PVLDB Reference Format:
Xu Chen, Haitian Chen, Zibo Liang, Shuncheng Liu, Jinghong Wang, Kai Zeng, Han Su, and Kai Zheng. LEON: A New Framework for ML-Aided Query Optimization. PVLDB, 16(9): XXX-XXX, 2023.
doi:XX.XX/XXX.XX

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

*The work was done during internship at Huawei Cloud GaussDB(DWS) Team.
†Corresponding authors

Proceedings of the VLDB Endowment, Vol. 16, No. 9 ISSN 2150-8097.
doi:XX.XX/XXX.XX

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 [34], query optimizer still struggles to deliver satisfactory performance and is time-consuming to maintain [16].

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 [3, 17, 20, 35, 50]. 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 [15, 22, 23, 46]. However, none of the "replacement" methods have been applied the practical usage. Commercial suppliers are still hesitant to incorporate them into their DBMSes.

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 omnipotent. 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 that has been studied for more than three decades [34]. 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 [7, 21].

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 [22, 23], 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 many 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 [4, 13, 16, 21, 36, 37]. 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.

Our approach: Based on the design principles, we propose a different framework for leveraging ML to aid the query optimizer—*LEON*. Specifically, we leverage the power of ML to make the traditional query optimizer start from 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 ML 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. And we only use ML to enhance the cost model since we find it is more effective compared to other factors, which will be further illustrated in Sec. 2.3 and Sec. 6.5. We design a mixed cost estimation combining both the expert cost model and the ML model to guide the plan search process. *LEON* takes the cost model as the initial cost estimation and use an ML model for cost calibration, to make the cost model more consistent with the user-defined goal (e.g., latency). The ML 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. 6.6, 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 [36, 37]. However, the widely used accuracy metric for cost estimation cannot reflect a method's end-to-end query performance [29], which prevents them from deployment in DBMSes. Intuitively, regardless of the absolute value, the plan decision is only based on the ranking of the candidate 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 [8, 48]. 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 ML 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 quantify the uncertainty of ML 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 [21, 22, 42]. 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 ML 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 ML 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.

Query optimization efficiency is an important criterion that we believe for a learned query optimizer in practical usage. The plan searching method in the traditional query optimizer is typically dynamic programming (DP) [34]. One of the main concerns for the ML-aided query optimizer is that ML models bring extra computation overhead to query optimization since the ML model needs to evaluate the candidate plans. To improve the plan search process, we use ML model-guided pruning strategy during searching for the optimal plans. The key idea is that many inferior subplans during plan search cannot be extended to the complete plan so the ML model can learn to prune the redundant search space based on the

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 [15, 22, 23, 42, 46]	**KnobTuner** [4, 18, 21, 47, 49]	**CardEst** [13, 22, 25, 26, 30, 41, 43]	**CostEst** [24, 36, 37]	**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

overall ranking for the complete plan. With the ML model guidance, we find that the search space can be pruned safely without hurting too much performance.

We conduct extensive experiments on the complex public benchmark. In our evaluation, *LEON* achieves state-of-the-art (SOTA) performance in terms of training efficiency and latency performance. *LEON* outperforms the expert DBMS consistently with 3-hours training, which is around 2× speedup compared to the SOTA RL-based optimizer. In addition, *LEON* shows a much more stable performance on both single query performance and average performance compared to previous works (about 8× less performance deterioration compared to other learning-based methods).

In summary, we make the following contributions:

- We analyze the fundamental limitations of ML for query optimization and point out design principles for ML-aided query optimizer (in Sec. 2).
- We present *LEON*, an ML-aided learning framework for the expert query optimizer. LEON integrates ML models deeply into traditional optimizers, combining both ML and expert knowledge. (in Sec. 3).
- We propose a contextual pairwise ranking objective for query optimization, which aims to help the ML-aided optimizer to learn better decisions (in Sec. 4).
- We propose a robust plan exploration strategy to help ML-aided optimizers improve performance (in Sec. 5.1).
- We introduce an ML model-guided pruning to improve the planning efficiency (in Sec.4.3).
- We conduct extensive experiments on benchmarks, demonstrating the superiority and practicality of *LEON* (in Sec. 6). We also integrate *LEON* into GaussDB(DWS).

2 BACKGROUND AND MOTIVATION

In this section, we first describe the traditional query optimizer. Then we analyze how ML methods tackle query optimization and compared them to traditional query optimizers. We summarize the existing learning-based query optimization methods and corresponding characteristics in Table 1. In general, we can categorize them into two classes: *ML-replaced* and *ML-aided*.

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 [11, 12, 34]. 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 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. 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 $Rel(q)$). 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 equivalent set S is chosen and memorized in the look-up table. (4) If $i = |Rel(q)|$, the search process finishes, and the optimal plan is returned. Otherwise, go back to step (1).

There is no heuristic involved in the plan enumeration. Theoretically, as long as the cardinality estimations and the cost model are accurate, this architecture obtains the optimal query 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 [15, 22, 23, 42, 46]. 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

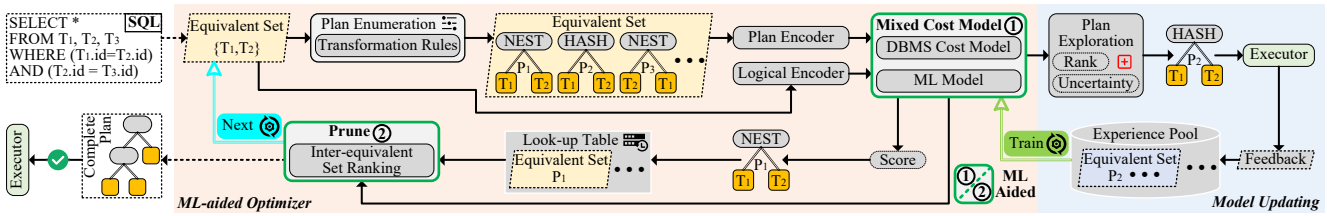

Figure 1: An overview of the ML-aided query optimizer. The optimizer maintains the DB knowledge and self-adjusts towards a database instance by ML models.

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 IO 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 [22] collects expertise experience from a traditional query optimizer. RTOS [46] pre-trains the ML model by cost as supervision. 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 shortcoming 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.

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 [4, 18, 47] 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 [21], 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: it only has coarse-grained optimization choices (enabling/disabling operators for the entire plan). For example, a subplan is optimal with a nested loop join but it can be discarded because a nested loop join has a poor performance on the other irrelevant 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 while-box methods include learned cardinality estimation (CardEst) [13, 22, 25, 26, 30, 41, 43] and learned cost estimation (CostEst) [24, 36, 37]. 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.

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.

3 FRAMEWORK OVERVIEW

In this section, we introduce a new framework for ML-aided query optimization called *LEON*. *LEON* trains an ML model to aid the expert query optimizer tailored to the specific database instance (dataset, workload, and hardware). The workflow of *LEON* is shown in Fig. 1. We first introduce how *LEON* utilizes an ML model to aid an expert query optimizer and then we describe how to train ML models.

ML-aided Query Optimizer. The standard search process is described in Section 2.1. ML model influences a standard query optimizer's optimization decision in the following two aspects (green box ① and ② in Fig 1).

① The first aspect is for intra-equivalent set optimization decisions (cost computation and cost comparison in Section 2.1). Instead of relying on the expert cost model, *LEON* learns a score function

$$M^I : (LF, PF) \rightarrow (\text{score, uncertainty}).$$

M^I maps a (logical feature (LF), physcial 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 lowest estimated score from M^I. Note that M^I also computes the uncertainty of the score (details in Section 5.1). The uncertainty indicates how confident is the model M^I to the predicted ranking position, which is aimed at plan exploration. M^I can easily disable the exploration mode during the online inference.

② The second aspect is for inter-equivalent set pruning. *LEON* learns a second score function:

$$M^O : (LF, PF) \rightarrow \text{overall score}.$$

Given $LF = (Q, q)$ and $PF = (\omega, p)$, M^O predicts an *overall* ranking of executing the complete query Q when the (sub)plan p is used as a partial step. Then, the equivalent set S with an inferior ranking position will be pruned from the look-up table.

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- or inter-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 ($M^I(LF, PF) \approx C(p)$ at the beginning). This approach solves the cold start problem as it avoids the need to collect a large amount of data when the ML model is unstable and reduces the risk of unexpected performance regression during the learning process. We call M^I_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. We discuss the configuration and training of the score function as follows.

ML Model. *LEON* trains a neural network with parameter θ to approximate the optimal score function M^I_θ and M^O_θ. Note that M^I_θ and M^O_θ use the same backbone network but only different prediction heads. The inputs to the neural networks 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).

Model Updating. For the task of updating ML model, *LEON* collect experience and use 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* will collect extra experience. After the plan search, *LEON* train ML models with collected experience. The following two steps (a) experience collection (details in Section 5.1) and (b) model training (details in Section 5.2) alternate until the predefined stopping condition. The workflow of model updating is illustrated in Fig 1.

(a) Experience Collection. *LEON* maintains an *experience pool* $E = \{(q, Q, p, \omega, C(p), L(p), L^O(p))\}$ to collect execution feedback including logical expression q, complete query Q, plan p corresponding to q, physical property ω, cost $C(p)$, immediate latency $L(p)$ and overall latency $L^O(p)$. 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-aid 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^I_θ to discover potentially better plans. The selected plans will be used to collect their execution feedback saved in E.

(b) Model Training. Model training aims to let the two ML models learn from the collected experience E. In the beginning, *LEON* initializes the mixed cost model M^I_θ 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 two models tailored to the goal (details in Section 4.1). For every iteration, *LEON* pick several batches of plan pairs (p_1, p_2) in experience pool E under certain contexts for the ML model. Then, *LEON* trains two ML models M_θ with standard supervised learning fashion by our proposed contextual ranking objective and safety regularization serving as the loss function. ML

models are trained on the collected experience iteratively to approximate the optimal score function.

4 PLAN RANKING MODEL

In this section, we first formalize the query optimization problem as a pair-wise classification problem in Section 4.1. Then we describe the key aspects to build an effective ML model in general. After that, we describe how to use *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 an ML model to predict the plan performance in a supervised regression manner, i.e., learning the exact latency. However, in practice, such learning objective has significant prediction errors [1].

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

Definition 1 (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. For inter-equivalent set pruning, context is defined as restrictions on the same Q but different on equivalent set ($S = (q, \omega)$). The context restriction helps *LEON* reduce unnecessary plan comparison.

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

Definition 2 (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 [8, 48]. In addition, the pairwise classification formulation is a more feasible way to train the ML model as we only need two labels to train the model instead of the whole ranking list.

4.2 ML Model

Here we describe the details of how to build appropriate ML models. The model architecture is shown in Fig. 2. The input to the

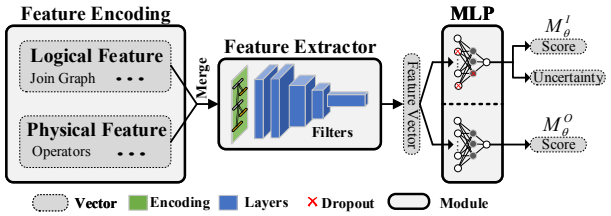

Figure 2: ML Model Architecture.

feature extractor contains two aspects: logical features and physical features. The logical features is 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 final encoding. Pattern matching for plan trees is commonly used and empirically verified by previous works [21, 22, 42]. *LEON* builds two ML models tasked to learn such patterns. M^O_θ and M^I_θ share the same backbone network called feature extractor. 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 [22] (denoted as filters in Fig. 2). 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.

Although the two models share part of the parameters, they are trained by different signals under different contexts in a pairwise manner. We defer the details of pairwise training to Section 5.2. Next, we introduce the differences between the two models.

Intra-equivalent set Model. *LEON* trains an intra-equivalent set ML model M^I_θ to influence the optimization decisions within an equivalent set. Based on Definition 2, the intra-equivalent set ML model training is supervised by the immediate latency performance of plan pairs from the same S. However, learning the parameterized cost model M^I_θ from scratch can be hard considering M^I_θ 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^I_θ by applying a parameterized calibration function $g_\theta(\cdot,\cdot)$ to the traditional cost estimation $C(\cdot)$:

$$M^I_\theta(LF, PF) = g_\theta(LF, PF)C(p),$$

where g_ϕ maps logical and physical features to a calibration ratio. 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^I_θ to introduce randomness (red cross in Fig 2). Thus, M^I_θ can measure uncertainty based on multiple predictions. The rationale will be illustrated in Section 5.1.

Inter-equivalent set Model. *LEON* trains another ML model M^O_θ for inter-equivalent set ranking. Different from M^I_θ, M^O_θ is used to prune the redundant search space. Thus, M^O_θ needs to identify the

inferior plans. M_θ^O is supervised by the overall latency signal of executing the complete query Q when using the current (sub)plan p as a partial step. Note that M_θ^O is trained in a pairwise fashion, however, with plan pairs from different equivalent sets.

4.3 Using *LEON* to Aid Query Optimizer

LEON, as a white-box method, deeply integrates the ML model into the DP. It ensures to use of *complete* plan space as the traditional query optimizer does and enhances the cost model to give accurate ranking to the candidate plans. *LEON* learns a ranking-based model within the context: intra-equivalent set and inter-equivalent set, which eliminates unnecessary plan comparison and model training in vast plan space. These advantages ensure that *LEON* employs a ranking-based ML model that is significantly more accurate than a regression model and expert cost model.

ML Model Integration into DP. *LEON* integrates ML models into DP search including two parts: the intra-equivalent set model M_θ^I and the inter-equivalent set model M_θ^O. M_θ^I is used to change the optimization decisions of the query optimizer in the equivalent set (under context $\forall p_1, p_2, S_1 = S_2, Q_1 = Q_2$). The usage is illustrated in Section 3. Here we focus on the usage of the inter-equivalent set model M_θ^O, which is used to improve the planning efficiency by pruning inter-equivalent set decisions.

M_θ^O is used to prune the inferior equivalent sets with the same number of join tables (in the same DP level). Specifically, after the cost comparison is finished for the same level, all equivalent sets memorize corresponding optimal plans. We use M_θ^O to evaluate all optimal plans for a level (under context $\forall p_1, p_2, S_1 \neq S_2, Q_1 = Q_2$) and prune $L\%$ last ranked equivalent sets. Those pruned plans will not be considered in the next level plan enumeration, thus it improves the inference efficiency dramatically. M_θ^O is used for pruning as it intuitively represents the overall performance for that subplan. Therefore, we can prune it greedily based on the overall performance. The pruning percentage $L\%$ and corresponding effect will be further shown in Section 6.6.

5 MODEL UPDATING

Model updating consists of two steps: experience collection and model training. Two ML models are trained iteratively. For every iteration, the current ML-aided optimizer collects execution feedback to experience pool E with our plan exploration strategy. The two models are trained by learning from the pairwise samples in the experience pool to improve current ML-aided optimizer performance. Here we describe the details of two modules.

5.1 Experience Collection

The plan exploration inherently connects closely to the plan enumeration in DP since the mixed cost model M_θ^I is tasked to find the optimal plans among enumerated plans i.e., from the equivalent set S. We pick not only the optimal one but also sub-optimal plans from S to the experience E. Later, the collected training data will be executed to collect corresponding execution feedback denoted as $E = \{(q, Q, p, C(p), \omega, L(p), L^O(p))\}$ (L(p) for M_θ^I and $L^O(p)$ for M_θ^O) The plan exploration strategy specifies how to select the valuable training data from S into experience E.

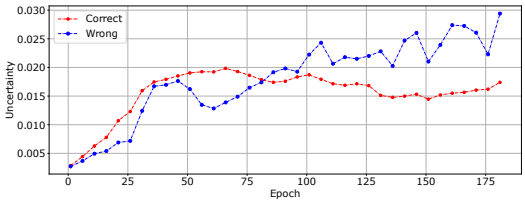

Figure 3: An empirical study on the relationship between uncertainty and wrong samples. Wrong samples are defined by the contextual pair-wise classification problem.

A unique challenge in query optimization is that it is time-consuming to get the execution feedback [21, 42]. *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) Top-k plans matter more. Normally, with more plans collected into E, the calibration model will perform better and lead to a better-calibrated cost model for query optimizer since the ML model can perform well with a large amount of data. However, collecting bad plans has little contribution to the mixed cost M_θ^I final decision since they can hardly influence the ranking of top-k candidate plans. This is intuitively correct because the higher-ranked plans hurt the ranking performance of the current model more than the lower-ranked ones.

(2) In addition, the ML-aided optimizer should correct its errors that lead to a sub-optimal query plan. Based on Definition 2, the error refers to the wrong ranking/classification of a plan pair p_1, p_2 enumerated from S. Ideally, if we know how confident a M_θ^I is to its estimation, the low-confidence predictions are more likely to make mistakes.

Model Uncertainty. Bayesian Neural Network (BNN) [14] provides a framework to measure the uncertainty of a predictive model. 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 by the posterior distribution $p(\theta' \mid E)$ for every θ:

$$p(\text{score} \mid LF, PF, E) = \int_\theta p(\text{score} \mid LF, PF, \theta') p(\theta' \mid E) d\theta'. \quad (1)$$

Based on Eq. (1), the model output can be approximated by $\mathbb{E}(\text{score} \mid LF, PF) \approx \frac{1}{N} \sum_{i=1}^{N} M_\theta^I(LF, PF)$, The uncertainty $u(LF, PF)$ for prediction (LF, PF) is defined as $u(LF, PF) = Var(\text{score}) \approx \sum_{i=1}^{N} M_\theta^I(LF, PF)^2 - \mathbb{E}(\text{score} \mid LF, PF)^2$. N is times of sampling the parameters from its posterior distribution. BNNs are data-efficient as it can learn from a limited data without overfitting [27].

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. 3 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 [8, 48] 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-k exploration and uncertainty-based exploration.

Stage 1: Top-k Exploration. At the first stage, we choose 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: Uncertain-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. When the ML 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 [14] by plugging dropout layers into M_θ^I. Based on Eq (1), we run M_θ^I 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.

5.2 Model Training

As described in Definition 2, we train the score function M_θ^I and M_θ^O in a supervised classification manner under different contexts. To train model M_θ^I, 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. Similarly, to train model M_θ^O, we pick several batches of plan pairs (p_1, p_2) satisfying $Q_1 = Q_2$ and $S_1 \neq S_2$. We assign correct label by $L(\cdot)^O$ instead of $L(\cdot)$. Next, we describe the loss function to train two ML models. Note that we use classification loss with regularization to train M_θ^I. We only use classification loss to train M_θ^O.

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

$$\mathcal{L}(\theta) = -(y \log(\sigma(LF_i, PF_i)) + (1-y)\log(1-\sigma(LF_i, PF_i)))$$
$$\text{where} \quad \sigma(LF_i, PF_i) = \frac{e^{-M_\theta(LF_i, PF_i)}}{\sum_{j=1}^{2} e^{-M_\theta(LF_i, PF_i)}} \quad for\ i = 1, 2. \tag{2}$$

σ is a *softmax* projection from the score to a probability to choose plan p_i and y is a true label by classifying a better plan in plan pair (p_1, p_2) based on the order of latency. 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 [45]. 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 [33].

6 EXPERIMENTS

6.1 Experiment Setup

Datasets. The four widely-used datasets listed below serve as benchmarks for the evaluation of *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 [22, 42]. 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 [21], 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 in total.

We employed varying levels of difficulty in our training/test spilt to validate the effectiveness of *LEON*. In the case of JOB and TPC-H benchmarks, we conducted evaluations under average circumstances. We randomly selected a query from each template to form the test set, while the remaining queries were utilized for the training set. Furthermore, for the JOB-EXT benchmark, we examined the generalization capability 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.

Baselines. The baselines are shown as below:

• **PostgreSQL [10].** 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 [42].** Balsa is a query optimizer based on reinforcement learning and learned models, and we utilize its source code [2] 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 configue Balsa in a non-parallel mode, with the same resource usage as other methods.

• **Bao [21].** Bao uses machine learning 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 [32] of Bao.

Expert Engines. For a fair comparison, all learning-based query optimization methods are implemented based on PostgreSQL. Similar to previous works [16, 42], 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 [46]: $GMRL = \prod_{i=1}^{n} \frac{Latency(q)}{Latency_{expert}(q)}$. 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.

Table 2: 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**

6.2 *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.

6.2.1 Overall Latency Performance After Training. Table 2 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.

LEON outperforms PostgreSQL, with speedups of 1.57×, 1.46×, 1.92×, 1.41× 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 1.14×, 1.41×, 1.33×, 1.08× speedup in terms of workload runtime on JOB, JOB-EXT, STACK, and TPC-H respectively. Compared with Bao, *LEON* achieves 1.21×, 1.23×, 1.35×, 1.34× 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.

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 6.2.2.

6.2.2 Query Regression Analysis. Fig. 4 presents the normalized runtime of each learning-based query optimizer (Balsa, Bao, 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, 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* exhibits a significant reduction in query performance regression 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* only causes 19% queries with performance regression. 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* 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 ML model learn faster and generate effective predictions.

6.2.3 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. 5 shows the training curve with 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.

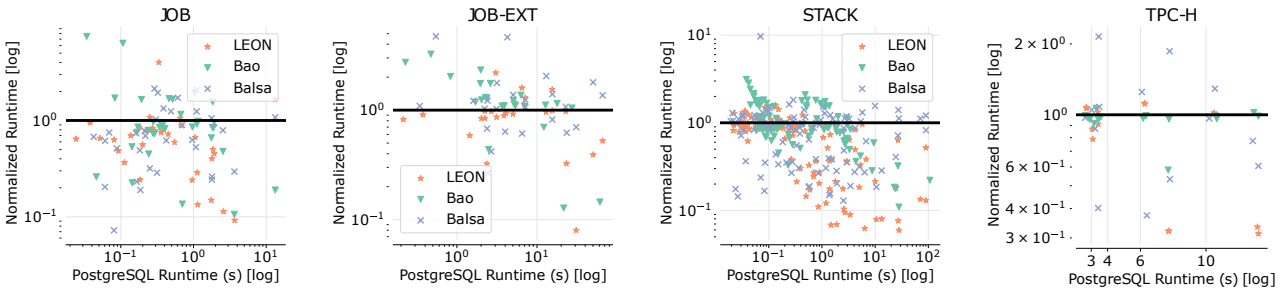

Figure 4: Breakdown of *LEON* 's per-query performance compared to the PostgreSQL runtime on different datasets.

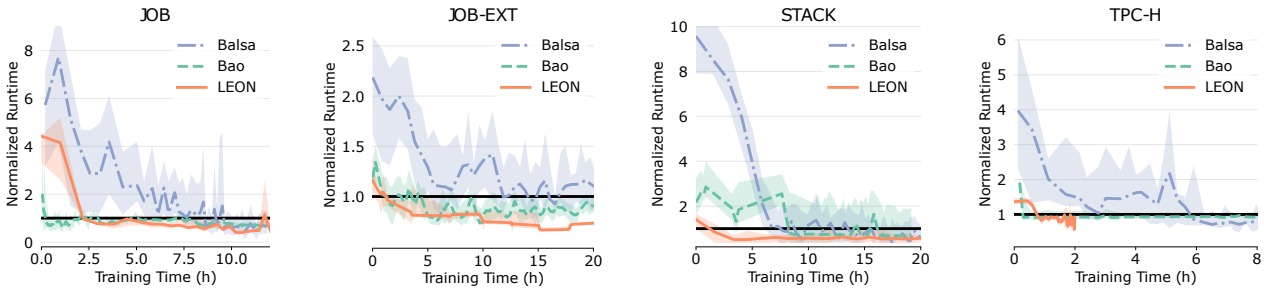

Figure 5: Training curves with variance on different datasets. The shaded area represents the range between the minimum and maximum values obtained from five different runs using different random seeds.

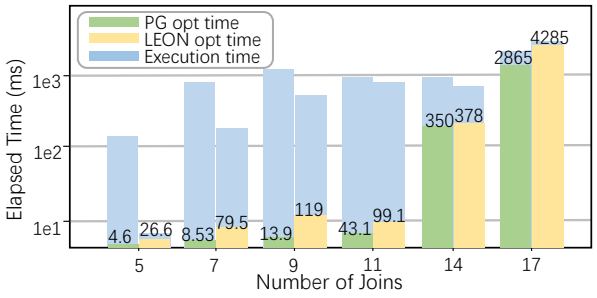

Figure 6: Breakdown of optimization and execution time on a different number of join tables.

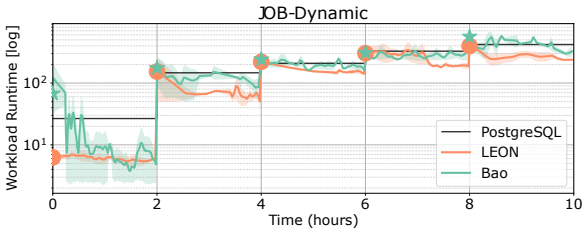

Figure 7: Training curve of dynamic workload.

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.

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.

6.3 Optimization time.

We conducted experiments on the JOB benchmark to compare the efficiency of *LEON* and PostgreSQL (PG) in terms of optimization (opt) time and execution time for a different number of joins. The results are presented in Fig. 6. Our findings suggest that *LEON* takes longer to optimize than PostgreSQL, mainly due to its modification of the PostgreSQL estimation and the additional computational overhead of evaluating candidate plans using the ML model. However, the ML-aided optimizer still achieves significant savings in

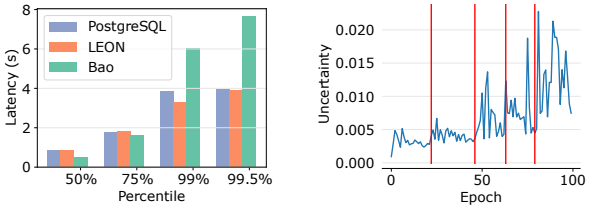

Figure 8: Tail latency and uncertainty for dynamic workload.

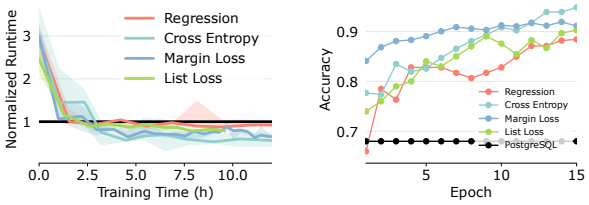

Figure 10: Impact of different ranking models.

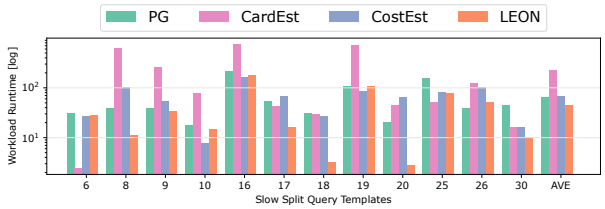

Figure 9: Workload Runtime of different templates on Slow-split JOB.

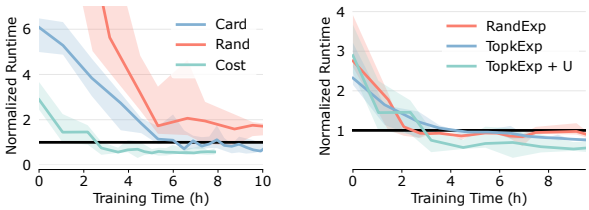

Figure 11: Impact of prior knowledge and impact of exploration strategies.

execution time, except for the 17-relation join, where both optimization time and execution time are comparable to PostgreSQL. Moreover, there is room for engineering optimizations in the implementation of *LEON*, which could further improve the performance of large queries. Overall, our results demonstrate that *LEON* can effectively find query plans that execute faster at a lower cost.

6.4 Adapting to Dynamic Workload

In this section, we demonstrate how *LEON* adapts to a dynamic workload during the course of training. We leverage the IMDB dataset and a dynamic workload generated from the original JOB workload. To showcase the change in workload, every two hours, we dynamically introduce two new templates to the current training and testing queries while keeping the previous queries. Each template contains ten training queries and ten distinct test queries. We continue this process until we have a total of 200 training queries and 200 testing queries, with join sizes ranging from 4 to 12. Fig 7 shows the training curse during the whole procedure. We only report Bao as a baseline because Balsa does not achieve competitive performance. We mark the initial performance of *LEON* and Bao at the beginning of each stage with dots and pentagrams, respectively.

Overall, *LEON* consistently outperforms PostgreSQL and Bao at every stage, with less initial performance regression. After about half an hour, *LEON* converges and continues to outperform the other methods due to its white-box approach that allows for a more complete search space without the restriction of a hint set. As more queries are accumulated, Bao shows larger performance regression, particularly at the beginning of stage five. In contrast, *LEON* exhibits less regression due to its ranking-based enhancement of the expert cost model, which is more accurate than a regression model. This experiment clearly demonstrates the adaptability of *LEON* to dynamic workloads.

We further compare the tail latency of different methods during the *whole* process. It can be shown from Fig 8 (left) that *LEON* outperforms PostgreSQL or remains stable at the 50%, 75%, 99%, and 99.5% percentiles. However, Bao exhibits performance regression at the 99% and 99.5% percentiles. It shows that Maintaining stability throughout the entire training process is a challenging task and stability of *LEON*.

Fig 8 (right) displays 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.

6.5 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 [44] as the CardEst work, and Tpool [37] as the CostEst. NeuroCard uses the IMDB dataset as training data, while we have used training queries provided by [37] for Tpool. We have injected their predicted cardinality and cost into PostgreSQL (PG) DP search to measure end-to-end latency performance. Similar to [42], we have split the slowest queries from JOB as the test workload. Fig 9 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 performance of ML models. 2) CardEst methods can be influenced by PG's inaccurate cost model, which has been studied in [16].

6.6 Analysis of design choices

Differet 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 [19] and Cross Entropy as two pairwise methods, and List Loss [40] as a listwise method. Fig 10 (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 10 (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.

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 this three initialization for ML models. In the case of Fig. 11 (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. 11 (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.

Pruning ratio. Tab. 3 shows the results of our analysis of the pruning ratio, as can be seen from the table. When the pruning ratio is 30%, the model has the lowest GMRL, the longest optimization time, and the most searched plans. This is because as the pruning ratio decreases, the number of execution plans that can be searched increases. It takes longer to optimize for selection, but a better execution plan can be found.

Table 3: Results of Different Pruning Ratios

Pruning ratio	Optimization time(s)	GMRL	Number of plans
70%	0.2389	0.7145	352
50%	0.4125	0.6513	778
30%	0.6742	0.5827	1026

7 RELATED WORK

ML-aided query optimizer. Leo [25] 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 [39, 41, 43, 51] and query-driven methods like [9, 29] are proposed to solve the cardinality estimation. Deep neural networks [36, 37] are trained in a supervised learning fashion to resolve cost estimation. Reinforcement learning (RL) helps solve decision-making problems such as database tuning problems [18, 38, 47]. Bao and its following works [21, 28, 49] propose to tune hint sets for each query, which is promising for practical usage.

Learned query optimizer. Recently, RL is applied to learn an optimizer to generate query execution plans. Neo [22] 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 [23], DQ [15] and RTOS [46] 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 [31] that change representations in the output layer. Balsa [42] shows the insight of learning an optimizer without the expert and achieves SOTA performance.

8 CONCLUSION

In this paper, we propose *LEON*, a framework for ML-aided expert optimization. Different from the existing learning-based methods, *LEON* train an ML 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. We conducted extensive experiments on four public benchmarks, providing evidence that *LEON* exhibits superior performance in terms of execution latency, training efficiency, and stability.

ACKNOWLEDGMENTS

This work is partially supported by NSFC (No. 61972069, 61836007, 61832017, 62272086), Shenzhen Municipal Science and Technology R&D Funding Basic Research Program (JCYJ20210324133607021), Municipal Government of Quzhou under Grant No. 2022D037, and Key Laboratory of Data Intelligence and Cognitive Computing, Longhua District, Shenzhen.

REFERENCES

[1] Mert Akdere, Ugur Çetintemel, Matteo Riondato, Eli Upfal, and Stanley B Zdonik. 2012. Learning-based query performance modeling and prediction. In *2012 IEEE 28th International Conference on Data Engineering*. IEEE, 390–401.

[2] balsa project. 2022. Balsa source code. https://github.com/balsa-project/balsa.

[3] Debabrota Basu, Qian Lin, Weidong Chen, Hoang Tam Vo, Zihong Yuan, Pierre Senellart, and Stéphane Bressan. 2015. Cost-model oblivious database tuning with reinforcement learning. In *Database and Expert Systems Applications*.

[4] Debabrota Basu, Qian Lin, Weidong Chen, Hoang Tam Vo, Zihong Yuan, Pierre Senellart, and Stéphane Bressan. 2016. Regularized cost-model oblivious database tuning with reinforcement learning. In *Transactions on Large-Scale Data-and Knowledge-Centered Systems XXVIII*. Springer, 96–132.

[5] Homanga Bharadhwaj. 2019. Meta-learning for user cold-start recommendation. In *2019 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 1–8.

[6] Zhe Cao, Tao Qin, Tie-Yan Liu, Ming-Feng Tsai, and Hang Li. 2007. Learning to rank: from pairwise approach to listwise approach. In *Proceedings of the 24th international conference on Machine learning*. 129–136.

[7] Bailu Ding, Sudipto Das, Ryan Marcus, Wentao Wu, Surajit Chaudhuri, and Vivek R Narasayya. 2019. Ai meets ai: Leveraging query executions to improve index recommendations. In *Proceedings of the 2019 International Conference on Management of Data*. 1241–1258.

[8] Jingtao Ding, Yuhan Quan, Quanming Yao, Yong Li, and Depeng Jin. 2020. Simplify and robustify negative sampling for implicit collaborative filtering. *Advances in Neural Information Processing Systems* 33 (2020), 1094–1105.

[9] Anshuman Dutt, Chi Wang, Azade Nazi, Srikanth Kandula, Vivek Narasayya, and Surajit Chaudhuri. 2019. Selectivity estimation for range predicates using lightweight models. *Proceedings of the VLDB Endowment* (2019).

[10] Lutz Fröhlich. 2022. PostgreSQL.

[11] Goetz Graefe. 1995. The cascades framework for query optimization. *IEEE Data Eng. Bull.* 18, 3 (1995), 19–29.

[12] Goetz Graefe and William J McKenna. 1993. The volcano optimizer generator: Extensibility and efficient search. In *Proceedings of IEEE 9th international conference on data engineering*. IEEE, 209–218.

[13] Yuxing Han, Ziniu Wu, Peizhi Wu, Rong Zhu, Jingyi Yang, Liang Wei Tan, Kai Zeng, Gao Cong, Yanzhao Qin, Andreas Pfadler, et al. 2021. Cardinality Estimation in DBMS: A Comprehensive Benchmark Evaluation. *arXiv preprint arXiv:2109.05877* (2021).

[14] Laurent Valentin Jospin, Hamid Laga, Farid Boussaid, Wray Buntine, and Mohammed Bennamoun. 2022. Hands-on Bayesian neural networks—A tutorial for deep learning users. *IEEE Computational Intelligence Magazine* 17, 2 (2022).

[15] Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph Hellerstein, and Ion Stoica. 2018. Learning to optimize join queries with deep reinforcement learning. *arXiv preprint arXiv:1808.03196* (2018).

[16] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How good are query optimizers, really? *Proceedings of the VLDB Endowment* 9, 3 (2015), 204–215.

[17] Guoliang Li, Xuanhe Zhou, and Lei Cao. 2021. AI meets database: AI4DB and DB4AI. In *Proceedings of the 2021 International Conference on Management of Data*. 2859–2866.

[18] Guoliang Li, Xuanhe Zhou, Shifu Li, and Bo Gao. 2019. Qtune: A query-aware database tuning system with deep reinforcement learning. *Proceedings of the VLDB Endowment* (2019).

[19] Lihao Liu, Qi Dou, Hao Chen, Jing Qin, and Pheng-Ann Heng. 2019. Multi-task deep model with margin ranking loss for lung nodule analysis. *IEEE transactions on medical imaging* 39, 3 (2019), 718–728.

[20] Lin Ma, Bailu Ding, Sudipto Das, and Adith Swaminathan. 2020. Active learning for ML enhanced database systems. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*.

[21] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2022. Bao: Making learned query optimization practical. *ACM SIGMOD Record* 51, 1 (2022), 6–13.

[22] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: A learned query optimizer. *arXiv preprint arXiv:1904.03711* (2019).

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

[24] Ryan Marcus and Olga Papaemmanouil. 2019. Plan-structured deep neural network models for query performance prediction. *arXiv:1902.00132* (2019).

[25] Volker Markl, Guy M Lohman, and Vijayshankar Raman. 2003. LEO: An autonomic query optimizer for DB2. *IBM Systems Journal* (2003).

[26] Guido Moerkotte, Thomas Neumann, and Gabriele Steidl. 2009. Preventing bad plans by bounding the impact of cardinality estimation errors. *Proceedings of the VLDB Endowment* (2009).

[27] Vikram Mullachery, Aniruddh Khera, and Amir Husain. 2018. Bayesian neural networks. *arXiv preprint arXiv:1801.07710* (2018).

[28] Parimarjan Negi, Matteo Interlandi, Ryan Marcus, Mohammad Alizadeh, Tim Kraska, Marc Friedman, and Alekh Jindal. 2021. Steering query optimizers: A practical take on big data workloads. In *Proceedings of the 2021 International Conference on Management of Data*.

[29] Parimarjan Negi, Ryan Marcus, Andreas Kipf, Hongzi Mao, Nesime Tatbul, Tim Kraska, and Mohammad Alizadeh. 2021. Flow-Loss: learning cardinality estimates that matter. *arXiv preprint arXiv:2101.04964* (2021).

[30] Jennifer Ortiz, Magdalena Balazinska, Johannes Gehrke, and S Sathiya Keerthi. 2019. An empirical analysis of deep learning for cardinality estimation. *arXiv preprint arXiv:1905.06425* (2019).

[31] Sinno Jialin Pan and Qiang Yang. 2009. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering* (2009).

[32] RyanMarcus. 2021. BAO source code. https://github.com/learnedsystems/BaoForPostgreSQL.

[33] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. 2015. Trust region policy optimization. In *International conference on machine learning*. PMLR, 1889–1897.

[34] P Griffiths Selinger, Morton M Astrahan, Donald D Chamberlin, Raymond A Lorie, and Thomas G Price. 1979. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*. 23–34.

[35] Ankur Sharma, Felix Martin Schuhknecht, and Jens Dittrich. 2018. The case for automatic database administration using deep reinforcement learning. *arXiv preprint arXiv:1801.05643* (2018).

[36] Tarique Siddiqui, Alekh Jindal, Shi Qiao, Hiren Patel, and Wangchao Le. 2020. 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*.

[37] Ji Sun and Guoliang Li. 2019. An end-to-end learning-based cost estimator. *arXiv preprint arXiv:1906.02560* (2019).

[38] Dana Van Aken, Dongsheng Yang, Sebastien Brillard, Ari Fiorino, Bohan Zhang, Christian Bilien, and Andrew Pavlo. 2021. An inquiry into machine learning-based automatic configuration tuning services on real-world database management systems. *Proceedings of the VLDB Endowment* (2021).

[39] Jiayi Wang, Chengliang Chai, Jiabin Liu, and Guoliang Li. 2021. FACE: a normalizing flow based cardinality estimator. *Proceedings of the VLDB Endowment* (2021).

[40] Xinshao Wang, Yang Hua, Elyor Kodirov, Guosheng Hu, Romain Garnier, and Neil M Robertson. 2019. Ranked list loss for deep metric learning. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 5207–5216.

[41] Ziniu Wu and Amir Shaikhha. 2020. BayesCard: A Unified Bayesian Framework for Cardinality Estimation. *arXiv e-prints* (2020).

[42] Zongheng Yang, Wei-Lin Chiang, Sifei Luan, Gautam Mittal, Michael Luo, and Ion Stoica. 2022. Balsa: Learning a Query Optimizer Without Expert Demonstrations. *arXiv preprint arXiv:2201.01441* (2022).

[43] Zongheng Yang, Amog Kamsetty, Sifei Luan, Eric Liang, Yan Duan, Xi Chen, and Ion Stoica. 2020. NeuroCard: one cardinality estimator for all tables. *arXiv preprint arXiv:2006.08109* (2020).

[44] Zongheng Yang, Amog Kamsetty, Sifei Luan, Eric Liang, Yan Duan, Xi Chen, and Ion Stoica. 2021. NeuroCard: One Cardinality Estimator for All Tables. *Proceedings of the VLDB Endowment* 14, 1, 61–73.

[45] Xue Ying. 2019. An overview of overfitting and its solutions. In *Journal of physics: Conference series*, Vol. 1168. IOP Publishing, 022022.

[46] Xiang Yu, Guoliang Li, Chengliang Chai, and Nan Tang. 2020. Reinforcement learning with tree-lstm for join order selection. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*.

[47] Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jiashu Xing, Yangtao Wang, Tianheng Cheng, Li Liu, et al. 2019. An end-to-end automatic cloud database tuning system using deep reinforcement learning. In *Proceedings of the 2019 International Conference on Management of Data*.

[48] Weinan Zhang, Tianqi Chen, Jun Wang, and Yong Yu. 2013. 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*. 785–788.

[49] Wangda Zhang, Matteo Interlandi, Paul Mineiro, Shi Qiao, Nasim Ghazanfari, Karlen Lie, Marc Friedman, Rafah Hosn, Hiren Patel, and Alekh Jindal. 2022. Deploying a Steered Query Optimizer in Production at Microsoft. In *Proceedings of the 2022 International Conference on Management of Data*.

[50] Xuanhe Zhou, Chengliang Chai, Guoliang Li, and Ji Sun. 2020. Database meets artificial intelligence: A survey. *IEEE Transactions on Knowledge and Data Engineering* (2020).

[51] Rong Zhu, Ziniu Wu, Yuxing Han, Kai Zeng, Andreas Pfadler, Zhengping Qian, Jingren Zhou, and Bin Cui. 2020. FLAT: fast, lightweight and accurate method for cardinality estimation. *arXiv preprint arXiv:2011.09022* (2020).