Lero: A Learning-to-Rank Query Optimizer

Rong Zhu[1,#], Wei Chen[1,#], Bolin Ding[1], Xingguang Chen[1,2],
Andreas Pfadler[1], Ziniu Wu[3], Jingren Zhou[1,*]
[1] *Alibaba Group*, [2] *The Chinese University of Hong Kong*, [3] *Massachusetts Institute of Technology*
[1]{red.zr, wickeychen.cw, bolin.ding, andreaswernerrober, jingren.zhou}@alibaba-inc.com
[2]xgchen@link.cuhk.edu.hk [3]ziniuw@mit.edu

ABSTRACT

A recent line of works apply machine learning techniques to assist or rebuild cost-based query optimizers in DBMS. While exhibiting superiority in some benchmarks, their deficiencies, e.g., unstable performance, high training cost, and slow model updating, stem from the inherent hardness of predicting the cost or latency of execution plans using machine learning models. In this paper, we introduce a *learning-to-rank* query optimizer, called Lero, which builds on top of the native query optimizer and continuously learns to improve query optimization. The key observation is that the relative order or *rank* of plans, rather than the exact cost or latency, is sufficient for query optimization. Lero employs a *pairwise* approach to train a classifier to compare any two plans and tell which one is better. Such a binary classification task is much easier than the regression task to predict the cost or latency, in terms of model efficiency and effectiveness. Rather than building a learned optimizer from scratch, Lero is designed to leverage decades of wisdom of databases and improve the native optimizer. With its non-intrusive design, Lero can be implemented on top of any existing DBMS with minimum integration efforts. We implement Lero and demonstrate its outstanding performance using PostgreSQL. In our experiments, Lero achieves near optimal performance on several benchmarks. It reduces the execution time of the native PostgreSQL optimizer by up to 70% and other learned query optimizers by up to 37%. Meanwhile, Lero continuously learns and automatically adapts to query workloads and changes in data.

1 INTRODUCTION

Query optimizer plays one of the most significant roles in databases. It aims to select an efficient execution plan for each query. Improving its performance has been a longstanding problem. Traditional *cost-based* query optimizers [40] find the plan with the minimum estimated *cost*, which is a proxy of execution latency or other user-specified metrics about resource consumptions. Such cost models contain various formulas to approximate the actual execution latency, whose magic constant numbers are exhaustively and extensively tuned based on engineering practice. Recent works [32, 33, 51] refine traditional cost models and plan enumeration algorithms with machine learning techniques. Although some progress has been made, they still suffer from deficiencies caused by the intrinsically difficult latency prediction problem.

In this paper, we propose Lero, a *learning-to-rank* query optimizer which features a new lightweight *pairwise* machine learning model for query optimization. Lero adopts a non-intrusive design, with minimal modification to the existing system components in DBMSs. Instead of building from scratch, Lero is designed to leverage decades of wisdom of query optimizers without extra, potentially significant, cold-start learning costs. Lero can quickly improve quality of query optimization by judiciously exploring different optimization opportunities and *learning to rank* them more accurately.

1.1 From Heuristic-Based Costing Models to Machine Learning Models

Traditional cost-based query optimizers [40] have three major components: *cardinality estimator*, *cost model*, and *plan enumerator*. For an input query Q, cardinality estimator can be invoked to estimate the *cardinality*, i.e., the number of tuples in the output, for each sub-query of Q. The *cost* of a plan P (with physical operators, e.g., merge join and hash join) for the query Q is a proxy of latency or other user-specified metrics regarding the efficiency of executing P. The cost model $\mathsf{PlanCost}(P)$ estimates P's cost and is usually a function of estimated cardinalities of Q's sub-queries. The plan enumerator considers valid plans of Q in its search space and returns the one with the minimum estimated cost for execution.

Various heuristics were essential in developing these components. For example, independence between attributes across tables is assumed and utilized for estimating cardinalities of joins of multiple tables [24, 45]. Magic constant numbers are prevalent in cost models. They are often calibrated and tuned over years to ensure that the estimated cost matches the plan's performance well empirically, under certain system and hardware configurations though. It is realized that such heuristics are not always reliable for varying data distributions or system configurations. As a result, cost models may produce significant errors and the plan generated from the traditional query optimizer may have poor quality [11, 16, 24, 45].

It is a natural idea to develop machine learning models to replace traditional cardinality estimators, heuristic-based cost models, and plan enumerators. For example, there are works on learning cardinality estimators (refer to [16, 59] for a survey). For learning cost models, different model parameters, instead of *fixed* magic constant numbers in traditional cost models, can be learned from different datasets and workloads to enable finer-grained characterization of various data distributions and system configurations, thus providing instance-level optimization of each query. A recent line of works on *learned query optimizers* are built upon the above idea and demonstrate some promising results [32, 33, 51].

Neo [33] and Balsa [51] provide end-to-end *learned* solutions for query optimization. A *plan value function* $\mathsf{PlanVal}()$, inspired by the value networks in deep reinforcement learning, is learned to replace the traditional cost model PlanCost. For a partial plan P' of

The first two authors contribute equally to this paper.
* Corresponding author.

the query Q, the machine learning model PlanVal(P', Q) predicts the minimum latency of a complete plan P that contains P' as a sub-plan, with statistics and patterns about the tables, predicates, and joins involved in P' and Q as input features. With PlanVal, Neo and Balsa use their best-first search strategies to find the best plan with the minimum estimated latency.

Bao [32] learns to steer a native query optimizer. It tunes the native query optimizer with different sets of *hints* to generate a number of different candidate plans for the query Q. Each hint set forces/disables some operations, e.g., index scan or hash join, so the query optimizer may output a different, and possibly better, execution plan. Bao uses a machine learning model PlanVal(P, Q) to estimate the quality (e.g., latency) of each candidate plan P and select the best candidate for execution. Meanwhile, Bao periodically updates its model parameters based on execution statistics using Thompson sampling [43], as solving a contextual multi-armed bandits problem [56] to minimize the overall performance regret.

Although learned query optimizers exhibit superiority than traditional ones in some applications [37], their performance is far from satisfactory. They still suffer from three major deficiencies:

- **Unstable performance.** These learned models are easily to produce inaccurate latency estimates, which leads to sub-optimal plans. Sometimes, the performance regression is very significant. As shown in Section 6.2, their performance could even be worse than PostgreSQL's native query optimizer on TPC-H benchmark.
- **High learning cost.** The cost of learning a query optimizer includes both the cost of exploring and executing different query plans and the cost of model training. Some models [33, 51] require tens to hundreds of training iterations to coverage, and each iteration to execute the newly explored plans. For complex datasets, this would consume several days to even weeks.
- **Slow model updating.** Existing learned optimizers need to update their latency prediction models to fit dynamic data, which is very challenging especially considering that the latency of the same plan may vary on dynamic data. Thus, model updating in this case requires huge efforts (training data and cost), and easily leads to performance regression, as shown in Section 6.4.

We recognize that, the deficiencies of both traditional and newly proposed learned query optimizers stem from the notoriously difficult problem of predicting the execution latency or cost of a plan. The exact execution latency depends on numerous factors [24, 25, 34, 39, 49], e.g., underlying data distribution, workload patterns, and system environment. Training such a prediction model is a costly operation, which requires collecting a large volume of training data, by executing queries and measuring latency statistics, and a lengthy training process to explore the huge hypothesis space.

Moreover, cardinality and cost estimations are only partial factors for query optimization. No matter whether with the traditional calibrated estimators/models or with the learned models trained on previous statistics, improving the estimations' accuracy does not necessarily lead to improvement in query optimization.

Is it really necessary to predict the latency? We ask such a fundamental research question: *for the purpose of query optimization, do we really need to estimate/predict the execution latency (or any other performance-related metric) of every possible query plan?* With the goal of searching the best execution plan, training a machine learning model to predict the exact latency (cost) is an overkill.

1.2 A Learning-to-Rank Query Optimizer

In this paper, we introduce a *learning-to-rank* query optimizer, called Lero. In essence, what we need for query optimization is a *learned* oracle that is able to *rank* a set of candidate query plans with respect to their execution efficiency.

Looking back, traditional cost models are essentially "human learning" models, whose parameters, i.e., magic constant numbers, have been tuned with decades of engineering efforts. Under the learning-to-rank paradigm, traditional and learned cost models can be regarded as *pointwise* approaches [28], which outputs an ordinal score (i.e., estimated cost) for each plan to rank them. Lero adopts an effective pairwise learning-to-rank approach without discarding human wisdom in developing traditional cost models and query optimizers. We summarize our contributions as follows.

- Lero applies *learning-to-rank* paradigm to query optimization. Compared to previous proposals to learn cost models or cardinalities which are partial factors for query optimization, Lero directly ranks and learns to improve plans' quality. Lero is more effective in selecting plans with good quality.
- Lero adopts a *pairwise* approach which learns to compare two plans and predict the better one. Compared to other *pointwise* approaches, training such a binary classifier is often easier[19]. Lero has significantly lower training costs, requiring fewer training samples, and much less time to train. A pairwise comparison model is also used for index tuning in [10]. Lero implements a novel deep learning model which consists of a classifier over two plan embeddings with shared parameters. The model is able to capture the detailed structural properties for given plans and is proven more effective in the context of query optimization.
- Lero is able to *explore* and *learn* new plan space (and new optimization opportunities) more effectively. Instead of using query-level hints as previous proposals [32], Lero can be equipped with various strategies to adjust cardinality estimates at expression level to effectively consider diversified plans and prioritize exploring more promising ones.
- Lero *leverages* decades of wisdom of databases and query optimization, rather than building a brand new optimizer from scratch. Lero starts with the default behavior of the native query optimizer and gradually *learns* to improve. Therefore, Lero is guaranteed with a good initial quality and does not need a lengthy cold-start training process.
- Lero employs a *non-intrusive* design. It can be implemented on top of any existing DBMS and integrated seamlessly with the native query optimizer. To be specific, Lero takes advantage of public interfaces provided by most DBMSs and jointly works with the native query optimizer to improve optimization quality.

We implement Lero on PostgreSQL [2] and conduct extensive experiments on several benchmarks to evaluate its efficiency and effectiveness. Lero reduces the execution time of the native query optimizer in PostgreSQL by up to 70% and other proposed learned query optimizers by up to 37%. Experiments show that Lero adapts

faster with stable performance to various query workloads and dynamically changing data.

The rest of this paper is organized as follows. Section 2 outlines Lero's *pairwise learning-to-rank* system architecture and key components. Section 3 presents the design of Lero's comparator model and describes how to train and infer using the model. Section 4 describes how Lero explores different plans and new optimization opportunities, for the purposes of both plan selection and model training. We report detailed evaluation results in Section 6. We cover related work in Section 7 and conclude in Section 8.

2 SYSTEM OVERVIEW

Lero is a *learning-to-rank* query optimizer which continuously explores different query plans, observes their performance, and *learns* to *rank* them more accurately. Lero adopts a *pairwise* approach whose objective is to predict which of two plans is more efficient. Compared with other *learning-to-rank* approaches, e.g., pointwise and listwise [28], Lero's pairwise approach makes the best trade-off between model accuracy and learning efficiency for our task.

Figure 1 shows the overall architecture of Lero. For each input query Q, the ***plan explorer*** works with the native query optimizer to generate a number of potentially good and diversified *candidate plans* $P_1, P_2, \ldots, P_n$. The *pairwise* ***plan comparator model*** CmpPlan is then invoked to select the best plan P^* from the candidates for answering Q. In system background, the ***model trainer*** executes other candidate plans using idle workers whenever system resources become available, collects the latency information into the *runtime stats repository*, and continuously trains the comparator model CmpPlan and updates it periodically. Such design enables the comparator model to be continuously trained and become better over time without affecting normal database services. The three components are briefly introduced as follows.

Plan Comparator Model. Formally, let $\text{CmpPlan}(P_1, P_2)$ be an oracle comparing any two plans P_1 and P_2 of a query:

$$\text{CmpPlan}(P_1, P_2) = \begin{cases} 0 & \text{if Latency}(P_1) < \text{Latency}(P_2) \\ 1 & \text{if Latency}(P_1) > \text{Latency}(P_2) \end{cases}, \quad (1)$$

with ties broken arbitrarily. For ease of presentation, we focus on execution latency $\text{Latency}(P)$ as the performance metric in this paper, but our solution can be easily generalized to other metrics.

Our comparator model is to learn the above oracle CmpPlan. To be more specific, we organize the training datasets (e.g. from runtime stats repository) in the form of $(P_i, P_j, label)$ for all pairs of executed plans of each query where *label* indicates which plan in a pair is better. The learning goal is to fit the output of the oracle in Eq. (1). We also use CmpPlan to denote the *learned plan comparator model* (or *comparator* for short). With CmpPlan trained sufficiently, we choose the plan P_i among all the candidate plans which maximizes $\text{Wins}(P_i)$ (with CmpPlan in Eq. (1) as the learned comparator) to execute. The design details of comparator model CmpPlan, including training and inference techniques, are described in Section 3.

Plan Explorer. For a query, the plan explorer generates a variety of n candidate plans $P_1, P_2, \ldots, P_n$. While the best one chosen by the comparator is returned by the optimizer for query execution, the rest candidate plans are used by the model trainer to refine the model. Therefore, the candidate plans serve the purposes of both plan selection and model training, and they should: i) contain some truly good candidates (*although we do not have to know who they are*) and ii) be sufficiently diversified so that the model could learn to distinguish between good and bad plans.

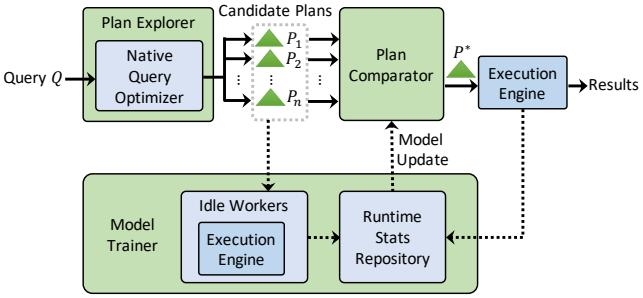


Figure 1: System architecture of Lero.

To satisfy the two requirements, our plan explorer uses the cardinality estimator as the tuning knob to generate more plans for each query: the cardinality estimates of sub-queries are purposely scaled up/down before being fed into the cost model for the native query optimizer to generate different candidate plans. In particular, it is shown that diversity can be introduced in the generated plans by tuning selectivities (cardinalities) of predicates in a query [9, 11]. Our plan explorer is built on the intuition that the plan from the native query optimizer is usually not too bad; by tuning cardinality estimates, it may either generate some better plans in the neighboring plan space (if tuning towards the true cost of a sub-query which was incorrectly estimated), or some worse plans (tuning in the opposite direction). Plans with various quality also increase the diversity among candidates to train the comparator, and it is the comparator's job to identify the best among them.

Details about our plan exploration strategy are described in Section 4. One obvious advantage of our plan explorer is that the cardinality estimator is an essential component in almost all query optimizers, and thus the strategy and the implementation of our plan explorer can be easily migrated to different databases.

Model Trainer. For each query, the trainer executes other candidate plans generated by the plan explorer, whenever system resources become available, and adds their plan information and execution statistics into the runtime stats repository. Such information is further used for training and updating the comparator model CmpPlan. By doing so, Lero is able to *explore* new plan space as much as possible and *learn* from its potential mistakes. In real-world distributed systems, idle workers and computation resources commonly exist due to scheduling [20] or synchronization [6]. Some platforms [4, 37] provide an individual environment for performance testing, which can be also used as a resource for executing candidate plans and collecting runtime stats.

3 A LEARNED PLAN COMPARATOR

The comparator CmpPlan is trained/updated to fit a training dataset from the runtime stats repository which continuously monitors query execution and collects execution information. For each incoming query, the plan explorer generates a list of candidate plans from which the best is picked for execution using CmpPlan.

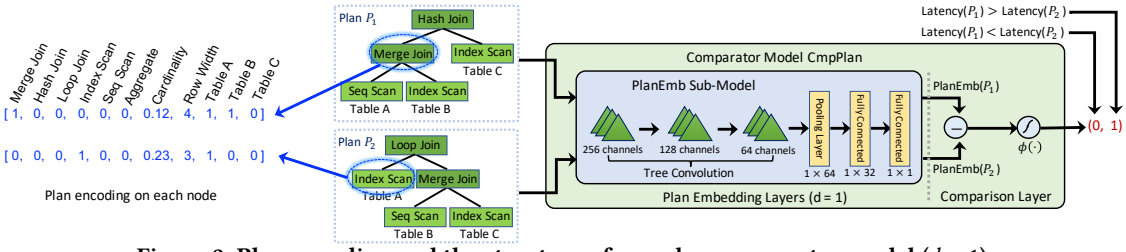


Figure 2: Plan encoding and the structure of our plan comparator model ($d = 1$).

3.1 Model Design

The overall model architecture of our comparator CmpPlan(P_1, P_2) is shown in Figure 2. It consists of *plan embedding layers* followed by a *comparison layer*. The plan embedding layer is carefully designed to effectively capture all the crucial information about a query plan, including tables, operators, and plan structural properties, etc., and the comparison layer combines features from the two plans and calculates their difference in terms of plan quality.

Plan Embedding Layers. The *plan embedding layers* of CmpPlan map P_1 and P_2 from the original feature space to a *one-dimensional (1-dim) embedding space*, in order to learn differences between plans. A sub-model PlanEmb takes features from each plan and generate its *plan embedding*, PlanEmb$(P_i) \in \mathbb{R}$ for each plan P_i $(i = 1, 2)$. We use the *parameter sharing* technique in machine learning: the two plan embeddings, PlanEmb(P_1) and PlanEmb(P_2), are generated by two copies of PlanEmb, which are two components in CmpPlan sharing the same model structure and learnable parameters.

Judicious design of embedding model to capture critical information in the tree-structured plan is absolutely crucial for the overall model's effectiveness and efficiency. Lero builds on top of the tree convolution model, similar to [32, 33, 36] but with significant improvement. As shown in Figure 2, each plan P is featurized as a tree structure of vectors. The vector for each sub-plan P_i that answers sub-query Q_i (corresponding to a node of the tree) concatenates: a one-hot encoding of the last operation on P_i, the (normalized) cardinality, the row width of Q_i's output, and 0/1 encoding of tables touched by Q_i. As the cardinality $\mathrm{C}(Q_i)$ spans in a wide range, we use a min-max normalization over $\log(\mathrm{C}(Q_i))$ in the feature vector. Unlike previous approaches [32, 33], the vector does not include the estimated cost, as it is strongly correlated with the estimated cardinality and row width and it might introduce additional inaccuracy of the cost model.

The tree convolution operation slides multiple triangle shaped filters over each node and its two children to transform the plan into another tree of the same size. Finally, vectors on the tree nodes are flattened to be fed into a multi-layer neural network to generate the plan embedding.

It is worth noting that previous works use machine learning models to predict latency. Our embedding model PlanEmb and comparator CmpPlan are trained and used differently. Instead of predicting plan latency, PlanEmb tries to extract key information from plans and enables the *comparison layer* of CmpPlan to compare two plans. The learned 1-*dim* plan embedding can be interpreted as a ranking criteria, and all pairs of plans for a query are comparable based on PlanEmb$(\cdot)$. Thus, the scale of PlanEmb(P) for a plan P is not restricted, and its value does *not* have to be proportional to (or approximate) P's performance metric, e.g., latency. Such flexibility allows the overall model to be more effective.

Comparison Layer. The plan embedding model PlanEmb$(\cdot)$ is learned together with CmpPlan via pairwise comparisons of plans and binary labels indicating which one is better for each pair. Thus, in the *comparison layer* of CmpPlan, we feed the difference $x = \mathrm{PlanEmb}(P_1) - \mathrm{PlanEmb}(P_2)$ of embeddings of two plans into a logistic activation function $\phi(x) = (1 + \exp(-x))^{-1}$ to generate the *model's final output* (indicating whether P_1 or P_2 is better):

$$\mathrm{CmpPlan}(P_1, P_2) = \phi(\mathrm{PlanEmb}(P_1) - \mathrm{PlanEmb}(P_2)), \quad (2)$$

which is within $(0, 1)$ and can be interpreted as how likely P_2 is more preferable than P_1. This is consistent with the learning goal of our comparator model as in Eq. (1): P_1 is more preferable if CmpPlan$(P_2, P_1) < 0.5$ or approaches to 0, and P_2 is more preferable if CmpPlan$(P_2, P_1) > 0.5$ or approaches to 1. Equivalently, a smaller 1-*dim* plan embedding is more preferable. It is obvious that

$$\mathrm{CmpPlan}(P_1, P_2) \to \begin{cases} 0 & \text{if } \mathrm{PlanEmb}(P_1, Q) \ll \mathrm{PlanEmb}(P_2, Q) \\ 1 & \text{if } \mathrm{PlanEmb}(P_1, Q) \gg \mathrm{PlanEmb}(P_2, Q) \end{cases}.$$

Our comparator model preserves two nice properties: i) (*commutativity*) CmpPlan$(P_1, P_2) = 1 - \mathrm{CmpPlan}(P_1, P_2)$, that is, exchanging the order of input plans does not affect the comparison result; ii) (*transitivity*): CmpPlan$(P_1, P_2) < 0.5$ and CmpPlan$(P_2, P_3) < 0.5$ $\Rightarrow$ CmpPlan$(P_1, P_3) < 0.5$, that is, P_1 is better than P_2 and P_2 is better than P_3 imply that P_1 is better than P_3. Therefore, the 1-*dim* plan embedding PlanEmb$(\cdot)$ induces a total order of all plans, and we could select $P^* = \arg\min_{P_1 \ldots P_n} \mathrm{PlanEmb}(P_i)$ with the minimum value of PlanEmb$(\cdot)$ as the best plan for execution.

Loss Function. The goal of CmpPlan is to maximize the likelihood of outputting the right order between any two plans, so that the best plan can be selected. Thus, the loss function of CmpPlan is designed towards this goal. Conceptually, let $\mathcal{A}$ be a (randomized) algorithm that decides which one of P_1 and P_2 is more preferable based on the model's output CmpPlan$(P_1, P_2) \in (0, 1)$:

$$\mathcal{A}(P_1, P_2) \to \begin{cases} P_1 & \text{with probability } 1 - \mathrm{CmpPlan}(P_1, P_2) \\ P_2 & \text{with probability } \mathrm{CmpPlan}(P_1, P_2) \end{cases}. \quad (3)$$

We have CmpPlan$(P_1, P_2) + \mathrm{CmpPlan}(P_2, P_1) = 1$ by our model construction (this is from how the model's output is derived in (2), and the fact that, for the logistic activation function $\phi(x)$, $\phi(a-b) + \phi(b-a) = 1$). Thus, the algorithm $\mathcal{A}$ is well-defined: $\mathcal{A}(P_1, P_2)$ and $\mathcal{A}(P_2, P_1)$ output P_1 with the same probability. Let $P_1 \prec P_2$ denote the event that Latency$(P_1) <$ Latency(P_2) and $P_1 \succ P_2$ vice versa. The probability that $\mathcal{A}$ makes the right decision (its accuracy) is:

$$\mathrm{ACC}(\mathcal{A}) = \Pr_{\substack{Q \sim \mathcal{Q} \\ (P_1, P_2) \sim \mathcal{P}(Q) \times \mathcal{P}(Q)}} \left[\mathbb{I}_{\mathcal{A}(P_1,P_2)=P_1} \cdot \mathbb{I}_{P_1 \prec P_2} + \mathbb{I}_{\mathcal{A}(P_1,P_2)=P_2} \cdot \mathbb{I}_{P_1 \succ P_2} = 1 \right],$$

where $\mathcal{Q}$ is the distribution of queries, $\mathcal{P}(Q)$ is the distribution of plans for a given query Q, and the indicator function $\mathbb{I}_x$ returns 1 if the condition x holds and 0 otherwise.

The goal is to train CmpPlan such that $\mathcal{A}$ is as accurate as possible. Suppose we have a workload of queries in $\mathcal{W} = \{Q\}$ and a set of candidate plans $P(Q) = \{P_1, \ldots, P_n\}$ for each query, as the training data. We use $\mathcal{A}$ to compare all pairs of candidate plans for each $Q \in \mathcal{W}$; the probability that $\mathcal{A}$ makes no mistake is:

$$\mathrm{ACC}(\mathcal{A}, \mathcal{W}) = \prod_{Q \in \mathcal{W}} \left(\prod_{P_i \prec P_j \in P(Q)} (1 - \mathrm{CmpPlan}(P_i, P_j)) \cdot \prod_{P_i \succ P_j \in P(Q)} \mathrm{CmpPlan}(P_i, P_j) \right). \quad (4)$$

We train CmpPlan to fit the observed orders of plans for queries in $\mathcal{W}$, so the loss function is chosen to be $L = -\log \mathrm{ACC}(\mathcal{A}, \mathcal{W})$:

$$L = -\sum_{Q \in \mathcal{W}} \sum_{P_i, P_j \in P(Q)} \left(\mathbb{I}_{P_i \prec P_j} \cdot \log(1 - \mathrm{CmpPlan}(P_i, P_j)) + \mathbb{I}_{P_i \succ P_j} \cdot \log(\mathrm{CmpPlan}(P_i, P_j)) \right), \quad (5)$$

which coincides with the *cross-entropy loss* in classification.

3.2 Model Training

Instead of training from scratch, we pre-train our comparator model CmpPlan offline first on synthetic workloads to inherit the wisdom of the native query optimizer and its cost model. It is then continuously trained and updated online on real workloads with training data collected during actual executions of query plans.

Model Pre-training: Starting from Traditional Wisdom. To leverage decades of wisdom of cost models developed in DBMSs, we pre-train the comparator model using the estimated costs of candidate plans (without executing them) generated from a sample workload of queries in an offline training stage. After pre-training, the 1-*dim* embedding PlanEmb(P) is expected to be an approximation to the native estimated cost PlanCost(P). Thus, the model is bootstrapped to perform similarly to the native query optimizer at the beginning. As more and more queries are executed, CmpPlan is continuously improved and tuned to adapt to possibly dynamic data distribution and workloads that drift over time.

In the native cost model, the cost of a plan is usually a piecewise linear or quadratic function of estimated cardinalities of sub-queries with magic constants as co-coefficients for different operators. The goal of pre-training PlanEmb(P) is to learn such functions in a data-agnostic way, so as to handle any unseen query. Thus, we could pre-train PlanEmb(P) purely using synthetic workloads. Namely, we randomly generate a number of plans with different join orders and predicates on different tables, and featurize them as the training data; we also randomly set the cardinality for each sub-plan of each plan and feed them into the native cost model (without executing the plans) to derive the estimated plan costs as labels. As the cost models are often in a class of functions with simple structures, the model pre-training converges very fast.

Pairwise Training. In the online stage, we train and update the comparator CmpPlan in the pairwise comparison framework. For each query Q, its candidate plans $\{P_1, \ldots, P_n\}$ are generated by the plan explorer to be introduced in Section 4; these plans are executed by idle workers, whenever system resources become available, with execution statistics collected in the runtime stats repository. According to the loss function in Eq. (5), we construct $n(n-1)$ training data points for each query: for each pair (i, j) with $1 \le i \ne j \le n$, we construct a data point with *features* (P_i, P_j) and *label* 1 if Latency(P_i) > Latency(P_j), and 0 otherwise.

Periodically, we use an SGD optimizer to update CmpPlan together with PlanEmb by backward propagation with the training dataset constructed above. For parameters in the sub-model PlanEmb, as they are shared in two copies of PlanEmb that output PlanEmb(P_i) and PlanEmb(P_j), respectively, we add up the gradients from the two copies for each shared parameter together to update this parameter in the model PlanEmb.

Our comparator, by nature of its design, adapts fast to dynamically changing data during the processing of model training and updating. When tuples are inserted or deleted into the database, the relative orders of two plans (i.e., labels of CmpPlan) are more robust than their execution latencies (i.e., labels of a latency prediction model). The latter would vary for the same plan even when tuples from the same distribution are inserted, which makes the training of a latency prediction model costly especially on dynamic data. Our experimental study in Section 6.4 verifies the above intuition.

4 PLAN EXPLORATION STRATEGY

The plan explorer of Lero generates a list of candidate plans $P_1, \ldots, P_n$ for a query Q, for two purposes. First, for the purpose of *query optimization*, Lero applies the comparator model to identify the best plan among the candidates for execution. Thus the candidate list must include some truly good plans for consideration.

Second, for the purpose of *plan exploration*, Lero *prioritizes* exploring other promising plans for the query. By executing them and comparing their performance with the chosen one, Lero is able to catch past optimization mistakes and adjust the model timely using the newly observed runtime information. In addition, whenever system resources are available, other candidate plans which are *diversified* sufficiently (e.g., with different join orders or in different shapes such as left-deep and bushy trees) are also considered so that Lero learns new plan space and improves the model over time.

For the ease of deployment, Lero makes minimal changes to the native query optimizer. Therefore, the plan explorer should be able to be implemented in a lightweight way, e.g., through re-implementing some system provided interfaces. Meanwhile, we hope that the framework of Lero could be generally used in any DBMS; thus, the strategy should not be system-specific.

Existing Plan Exploration Methods. A straightforward strategy is to explore a random sample of valid plans for each query, which is used by reinforcement learning-based approaches such as Neo [33] and Balsa [51]. The obvious drawback is that, with high likelihood, high-quality plans could be missing in a random sample; otherwise, the sample size has to be so large that executing these sample plans would be too costly for model training.

Bao [32] explores plans by tuning a set of hints (boolean flags) to disable/force certain types of optimization rules. For example, the native query optimizer initially generates a plan with merge join

for a query; with a hint set that disables merge join and force indexed nested loop join, it would generate a different candidate plan. However, this optimizer-level tuning strategy has two drawbacks.

First, a hint set is typically applied for the whole query during the entire plan search procedure. If different parts of a query have different optimal choices, tuning a flag at a query level may miss opportunities for finding high-quality plans. For example, let Q be a query joining two sub-queries Q_1 and Q_2 where the best join operations for Q_1 and Q_2 are indexed nested loop join and merge join, respectively. The query optimizer may use merge join for both Q_1 and Q_2 due to cardinality estimation errors. With a hint set disabling/forcing either merge join or indexed nested loop join, at most one of Q_1 and Q_2 could select the right operation. In such cases, the optimal plan can never be obtained by hint set tuning and included in the candidate list.

Second, the set of available hints is system specific. An optimizer usually contains hundreds of flags to enable/disable certain optimization rules. Enumerating all kinds of combinations is infeasible in practice. Selecting an effective subset of hint sets manually requires a deep understanding on the system and comprehensive analysis on the workload [37].

Thus, it motivates us to pursue other routes for designing a plan exploration strategy. We introduce our tuning knob in Section 4.1, followed with practical heuristic strategies in Section 4.2.

4.1 Cardinality as Knob for Plan Explorer

Lero uses the cardinality estimator as the tuning knob for our plan explorer. In the native cost-based query optimizer, the estimated cardinality for each sub-query of an input query Q is fed into the cost model to guide plan enumeration and selection. Each time with different cardinality estimates on one or more sub-queries, the query optimizer would select a different plan for Q. In Lero's plan explorer, we tune (magnify or reduce) the estimated cardinalities multiple times to generate a list of different candidate plans.

Formally, let $\mathrm{C}()$ be the native cardinality estimator in DBMS. For each sub-query Q' of a query Q, $\mathrm{C}(Q')$ gives a cardinality estimate. Instead of invoking $\mathrm{C}()$ in the cost model, we ask the query optimizer to invoke a *tuned estimator* $\widetilde{\mathrm{C}}()$, so it would generate a different plan. Our plan explorer would tune $\mathrm{C}()$ in different ways; with different tuned estimators fed into the cost model, the query optimizer generates different plans as the candidates $P_1, \ldots, P_n$.

Using the cardinality estimator as a tuning knob has following advantages. First, in cost-based query optimizers, cardinality estimates decide the estimated costs, and thus determine join orders and physical operations on tables and sub-queries. Therefore, under the same resource budget (e.g. work memory), tuning cardinality estimates would be highly possible to introduce diversity in candidate plans, possibly with different join orders or different operators (e.g., sort-merge join v.s. nested loop join). Second, cardinality tuning is platform-independent. For most DBMSs, there exist system-provided interfaces to modify the estimated cardinality, which is friendly for system deployment.

A Brute-Force Exploration Strategy. Let $\mathrm{C}^*()$ be the true cardinality. For any query Q, the difference between $\mathrm{C}^*(Q)$ and the DBMS's estimate $\mathrm{C}(Q)$ is unknown. However, with a reasonable number of different ways to tune $\mathrm{C}(Q)$, we can ensure that at least one tuned estimator $\widetilde{\mathrm{C}}(Q)$ is close to $\mathrm{C}^*(Q)$, in terms of *q-error*, which is defined as $\mathrm{QE}(\text{estimate}, \text{true}) = \max\{\frac{\text{estimate}}{\text{true}}, \frac{\text{true}}{\text{estimate}}\}$.

We tune the cardinality estimator $\mathrm{C}()$ with exponentially varying step sizes. Suppose we know an upper bound of the q-error of $\mathrm{C}()$, namely, $\mathrm{QE}(\mathrm{C}(Q'), \mathrm{C}^*(Q')) \le \Delta$ for any sub-query Q' of Q. Let

$$F_\alpha^\Delta = \{\alpha^t \mid \lfloor -\log_\alpha \Delta \rfloor \le t \le \lceil \log_\alpha \Delta \rceil, t \in \mathbb{Z}\} \tag{6}$$

be the set of *guesses of scaling factors*. For each $f = \alpha^t \in F_\alpha^\Delta$, we tune $\mathrm{C}(Q')$ as $\widetilde{\mathrm{C}}(Q') = f \cdot \mathrm{C}(Q')$. Then there is at least one $f \in F_\alpha^\Delta$ such that $\mathrm{QE}(f \cdot \mathrm{C}(Q'), \mathrm{C}^*(Q')) \le \alpha$. The estimation of Δ does not need to be tight, since the number of guesses $|F_\alpha^\Delta| = \Theta(\log_\alpha \Delta)$ depends logarithmically on Δ. In practice, it can be set based on users' experience or q-error distribution on historical queries.

In order to explore plans for the query Q, we repeat the above tuning process recursively for all of its sub-queries: for each sub-query Q' of Q, we pick a scaling factor $f_{Q'} \in F_\alpha^\Delta$ and set $\widetilde{\mathrm{C}}(Q') = f_{Q'} \cdot \mathrm{C}(Q')$. For each combination of scaling factors $\langle f_{Q'} \rangle_{Q' \subseteq Q}$, we construct an estimator $\widetilde{\mathrm{C}}()$ defined for all the sub-queries of Q, and we can feed it into the cost model to generate a candidate plan.

From the way how tuned estimators are constructed above, there is at least one combination of scaling factors such that the resulting $\widetilde{\mathrm{C}}()$ satisfies $\mathrm{QE}(\widetilde{\mathrm{C}}(Q'), \mathrm{C}^*(Q')) \le \alpha$ for all sub-queries of Q. Theoretically, a near-optimal candidate plan under a specific cost model can be generated by the optimizer using an estimator $\widetilde{\mathrm{C}}()$ with q-error bounded by α; more formally, from [35], with some mild assumptions about the cost model, the optimal plan under $\widetilde{\mathrm{C}}()$ is no worse than the optimal plan under $\mathrm{C}^*()$ by a factor of α^4.

Ensuring that at least one candidate plan is near-optimal suffices for the purpose of optimization, as it is the comparator's job to identify it among all candidates. However, the overhead of the above method is too high: the number of different sub-queries Q' is at least 2^q where q is the number of tables in Q, and $|F_\alpha^\Delta| = \Theta(\log_\alpha \Delta)$; thus, the total number of combinations of scaling factors is $\Theta(\log_\alpha^{2^q} \Delta)$. From the above discussion, we have the following results.

Proposition 1. *In the above brute-force exploration strategy, the number of candidate plans generated is at most* $\mathrm{O}(\log_\alpha^{2^q} \Delta)$*; with some mild assumptions about the cost model as in [35], at least one of them is no worse than the optimal plan by a factor of* α^4.

Thus, the list of candidate plans it generates is too long, which is very costly for model training. Since this method is only applicable for queries joining a very small number of tables, we propose more effective heuristic methods in the following subsection.

4.2 Priority-Based Heuristic Methods

Plan exploration is conducted in background whenever system resources become available. The key idea of our heuristics is to *prioritize* exploring where the native optimizer is likely to make mistakes. In addition, instead of exploring all possibilities and tuning cardinality estimates for all sub-queries simultaneously as in the brute-force strategy, we introduce a heuristic which focuses on mistakes in estimating cardinalities for size-k sub-queries on k tables (for each different $k \ge 1$ at one time). [1]

The necessity of considering $k > 1$ is because the difficulty (as well as the error) of estimating cardinality of size-k sub-queries

[1] We also consider heuristics based on plan diagram[9, 11], which can be found in the extended technical report [58].

Algorithm plan_explorer(Q, α, Δ)
1: Priority queue candidate_plans $\leftarrow \emptyset$
2: **for** each $f \in F_\alpha^\Delta$ in the increasing order of $|\log f|$ **do**
3: **for** $k \leftarrow 1$ to q (the number of tables in Q) **do**
4: **Inside query optimizer:** let $\mathrm{C}()$ be the default cardinality estimator in native query optimizer
5: $\widetilde{\mathrm{C}}(Q') \leftarrow f \cdot \mathrm{C}(Q')$ for size-k sub-queries $Q' \in \mathrm{sub}_k(Q)$
6: $\widetilde{\mathrm{C}}(Q') \leftarrow \mathrm{C}(Q')$ for sub-queries $Q' \in \mathrm{sub}(Q) - \mathrm{sub}_k(Q)$
7: feed cardinality $\widetilde{\mathrm{C}}()$ into the cost model to generate a plan P
8: candidate_plans $\leftarrow$ candidate_plans $\cup \{P\}$
9: **return** candidate_plans

Figure 3: Plan exploration strategy in Lero.

increases significantly as k increases (due to cross-table correlation among columns). Thus, the native optimizer tends to make mistakes in generating partial plans for sub-queries on a larger number $k > 1$ of tables. As a trade-off between efficiency and effectiveness, the following strategy tries to guess where the biggest mistake is (by enumerating all possible values of k), and tune cardinality estimates for all size-k sub-queries together.

The plan explorer in Lero is illustrated in Figure 3. In the outer loop (line 2), we enumerate a scaling factor $f \in F_\alpha^\Delta$, in the increasing order of $|\log f|$ with ties broken arbitrary. That is, we prioritize exploring the neighboring plan space surrounding the native optimizer's choice (with f closer to 1), where plan quality is not too bad and the optimizer is likely to make mistakes, if any. Let $\mathrm{sub}_k(Q)$ be the set of size-k sub-queries of a query Q on q tables. An extensive experimental study in [24] shows that sub-queries with the same size are often underestimated together with similar q-errors. Thus, in the inner loop, we enumerate k and tune the cardinality estimates for all sub-queries $Q' \in \mathrm{sub}_k(Q)$ together, resulting in a tuned estimator $\widetilde{\mathrm{C}}$ (lines 5-6) to be fed into the optimizer. The resulting candidate plans are maintained in a priority queue for evaluation and execution whenever system resources allow.

Proposition 2. *In the heuristic strategy in Figure 3, the number of candidate plans generated is at most* $\mathrm{O}(q \cdot \log_\alpha \Delta)$.

Encouraging Diversity in Candidates. An important goal of our plan explorer is to generate diversified candidates. In Figure 4, we give conceptual examples of typical cases when our plan explorer encourages diversity in the candidates (with different plan shapes and join orders), for a query joining four tables, $A \bowtie B \bowtie C \bowtie D$:

(*Diversity of plan shapes*) Two valid plans P_1 and P_2 have different shapes (left-deep tree and bushy tree, respectively). When we tune cardinality estimates for size-2 sub-queries ($k = 2$ in line 2 of Figure 3), different values of f would encourage different plan shapes. Namely, when $f \ll 1$, two sub-queries $A \bowtie B$ and $C \bowtie D$ in P_2 have reduced cardinality estimates in $\widetilde{\mathrm{C}}()$, while only one sub-query $A \bowtie C$ in P_1 has a reduced cardinality estimate. As a result, the estimated cost of P_2 is reduced more than that of P_1, and thus a bushy tree is more likely to be generated by our plan explorer and included in the candidate list. When $f \gg 1$, with a similar logic, a left-deep tree is more likely to appear in the candidate list.

(*Diversity of join orders*) Plans P_3 and P_4 are both left-deep trees but have different join orders. In Figure 4, the length of bar on each single and joined table represents its size. P_3 first executes $A \bowtie B$ which has a large estimated cardinality, while P_4 first executes $C \bowtie D$ which is estimated to be much smaller. When we tune cardinality estimates for size-2 sub-queries, we multiply both estimates with a scaling factor f. Suppose P_3 and P_4 have similar total estimated cost before tuning. When $f \ll 1$, the *decrease* of the tuned cardinality estimate $\widetilde{\mathrm{C}}(A \bowtie B) - \mathrm{C}(A \bowtie B)$ for $A \bowtie B$ is more significant than $\widetilde{\mathrm{C}}(C \bowtie D) - \mathrm{C}(C \bowtie D)$, and thus P_3 is more likely to be in the candidate list; when $f \gg 1$, the *increase* $\widetilde{\mathrm{C}}(A \bowtie B) - \mathrm{C}(A \bowtie B)$ is more significant, and thus P_4 is more more preferable.

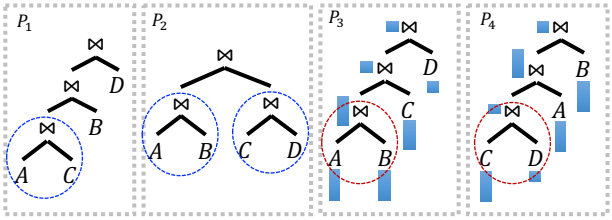


Figure 4: Intuitions on why our plan explorer based on sub-query grouping encourages diversity.

Some Implementation Details. The tuned estimator $\widetilde{\mathrm{C}}()$ in our plan explorer (in Figure 3) is not constructed explicitly by setting the cardinality estimate for every possible sub-query Q' of Q (as in lines 5-6). Instead, we only need a "hook" to the native cardinality estimator $\mathrm{C}()$: if the size of a sub-query Q' is k, we invoke $\mathrm{C}(Q')$ and return $f \cdot \mathrm{C}(Q')$ as $\widetilde{\mathrm{C}}(Q')$; otherwise, we just return $\mathrm{C}(Q')$. We implement Lero on PostgreSQL. Thanks to Lero's non-intrusive design, the implementation is straightforward. The only major modification to PostgreSQL is a hook function pg_hint_plan that implements the tuned cardinality estimator $\widetilde{\mathrm{C}}()$ in the above way. Our implementation on PostgreSQL can be easily ported to other databases that offer similar interfaces for cardinality estimators.

5 EXTENSIONS AND DISCUSSION

Comparator with d-Dimensional Embedding. Our comparator model CmpPlan could be extended to a d-dimensional (d-dim) embedding space. We only need to leave the dimensionality of the last plan embedding layer (in Figure 2) as d, with the hope to summarize more sophisticated statistical and structural information about each plan in plan embeddings. After that, the comparison layer is a learnable linear layer that compares the two d-dim embeddings $\mathrm{PlanEmb}(P_1)$ and $\mathrm{PlanEmb}(P_2)$, and outputs 0 (P_1 is better) or 1 (P_2 is better). We defer detailed analysis to the technical report [58].

How to Handle Varying Resource Budget. Traditionally, each query is assigned a resource budget (e.g. work memory) under which the query is optimized and later executed. It is the database engine's responsibility to guarantee each query receives its assigned budget at runtime and ensure performance isolation among different queries to avoid any resource contention. This is an orthogonal task to query optimization. When resource condition in the database engine changes dramatically and the assigned budget for a query can no longer be guaranteed at runtime, re-optimization for the query under a different budget might be triggered.

We assume constant resource budget in this paper. In principle, Lero can include resource budget information into the feature set of our comparator model, namely, as inputs to the plan embedding layers. In this way, the resulting plan embeddings encode such

information about available resource, and plans can be compared in the comparison layer under different resource budgets at runtime. This augmented design will bring more challenges in model training, as the comparator now needs to observe runtime stats under various resource budgets to be fully trained.

6 EXPERIMENTAL EVALUATION

We make our implementation of Lero open-source [2]. We also implement recently proposed learned query optimizers, Neo [33] and Bao [32], and use Balsa's open-source implementation in [1, 51] on PostgreSQL. We describe our experimental setup in Section 6.1. We first answer the most crucial questions about Lero's performance:

- How much improvement on query execution performance could Lero achieve in comparison with PostgreSQL's native and other learned query optimizers? (Sections 6.2 and 6.8)
- How much is Lero's query optimization cost? (Section 6.3)
- Could Lero adapt to workloads on dynamic data? (Section 6.4)

We then examine the design choices and settings in Lero and understand how they affect the performance of Lero:

- What are the benefits of using pre-training in Lero? (Section 6.5)
- Is the proposed plan exploration strategy efficient and effective? How is it compared with alternative strategies? (Section 6.6)
- With less or limited idle resource, could Lero still achieve similar performance gains? (Section 6.7)

6.1 Experimental Setup

Benchmarks. We evaluate all the query optimizers on three widely used benchmarks.

- The IMDB dataset has 21 tables on movies and actors, and its JOB workload [24] has 113 realistic queries. For large-scale evaluation in most of our experiments, we generate a workload of 1000 queries from JOB (similar to the one in [32]): each time we randomly sample a query template from JOB, fetch its join template and attach some randomly generated predicates to it.
- The STATS dataset and STATS-CEB workload [15, 16] are recently proposed to evaluate the end-to-end performance of query optimizer. STATS contains 8 tables of user-contributed content on the Stats Stack Exchange network. Its data distribution is more complex than IMDB. STATS-CEB contains 146 query templates varying in join sizes and types. We generate a query workload using the same approach as described above.
- The TPC-H [8] has its data synthetically generated under a uniform distribution. We set the scale factor to 10, and use its query templates #3, 5, 7, 8, 9, 10 for workload generation. Under each template, we generate a number of queries with varying predicates. We exclude other templates which are either too simple (on only one or two tables), or with views or nested SQL queries. Queries with views or nested SQL queries cannot be fully optimized by Lero due to a limitation of our implementation: to tune cardinality estimations for operators in the plans, we modify the hook function pg_hint_plan, which does not support to impose any hints on views and nested SQL queries. We would try to fix this limitation in the future implementation.
- The TPC-DS [7] is another benchmark for evaluating database performance. In similar to TPC-H, we also set its scale factor to 10 and exclude all templates that are either too simple or can not be supported by the pg_hint_plan hook function. The remaining templates are used to generate training and testing queries.

Learned Optimizers in Comparison. Neo and Balsa need to find a plan to execute using their latest model in each training epoch, and use the execution statistics to update the model. The generated plans are unknown before each epoch, so the training can only be done sequentially, which leads to a very long training time even with unlimited resource. By their evaluation [33, 51], the models converge after tens of epochs. On the contrary, both Bao and Lero can simultaneously run the selected plans and collect the training data in background, which greatly expedite the training process.

In our experiments, neither Neo nor Balsa could match the performance of PostgreSQL's native query optimizer after training for 72 hours on all of our datasets, except the original JOB workload (with only 113 queries) on the IMDB dataset. On the contrary, Bao and Lero often outperform PostgreSQL after training for several hours. Thus, we only report performance of Bao and Lero in most of our experiments in Sections 6.2-6.7. We compare Lero with Balsa on IMDB with the original JOB workload in Section 6.8. Bao and Balsa have demonstrated their superiority over Neo in [32, 51], so we do not further compare with it in the rest figures.

As described in [32], Bao selects one candidate plan by *Thompson sampling* to execute and collects its execution time to update its model periodically. To have a fair comparison with Lero, we also implement an extended version of Bao, called Bao+, which witnesses more plans for model training. For each training query, it runs all candidate plans generated by its hint set tuning strategy using idle computation resource and collects their execution time to update the model periodically, in a similar way to Lero.

Evaluation Scenarios We compare Lero with PostgreSQL's native query optimizer and other learned optimizers in different settings. PostgreSQL's native query optimizer does not need a separate training phase. For fair comparisons with the native and learned optimizers, we use the "time series split" strategy [32] for training and evaluating Bao, Bao+, and Lero. Namely, unique queries in a workload are randomly shuffled as $Q_1, Q_2, \ldots$. The learned optimizers are always evaluated on queries that have *never* seen during the procedure of model training and updating. In the experiments, we evaluate their performance under two realistic scenarios:

- *Performance curve since deployment.* The learned optimizers are continuously updated since deployed for each workload. Bao+ and Lero may execute multiple candidate plans for a query on idle workers. When Q_{t+1} is evaluated, all learned optimizers are only trained with information from earlier queries $Q_1, \ldots, Q_t$. Their models are updated in the background every 100 queries on IMDB and STATS, every 30 queries on TPC-H and every 50 queries on TPC-DS. By reporting the (accumulated) latency on each Q_{t+1}, we meter the performance of different optimizers since the deployment and how quickly they adapt to a new workload.
- *Performance with stable models.* The learned optimizers tend to be stable (that is, the model training process converges) after seeing a sufficient number of queries in the workload, i.e., $Q_1, \ldots, Q_T$

Query Optimizer	Execution Time (in hour)			
	STATS	IMDB	TPC-H	TPC-DS
PostgreSQL	20.19	1.15	0.94	1.68
Bao	15.32	0.47	1.17	1.55
Bao+	13.85	0.41	0.89	1.57
Lero	**11.32**	**0.35**	**0.74**	**1.47**
Fastest Found Plan	**10.73**	**0.19**	**0.72**	**1.39**

Table 1: Overall performance of different query optimizers.

(*training queries*), then we would use the optimizers to evaluate queries Q_{T+1}, Q_{T+2}, ... (*test queries*) without further updating the models. By reporting their (average) performance on Q_{T+1}, Q_{T+2}, ..., we compare the performance of learned optimizers after they are deployed on a workload and stabilized for a while.

Setup. We deploy learned optimizers on a Linux machine with an Intel(R) Xeon(R) Platinum 8163 CPU running at 2.5 GHz, 96 cores, 512GB DDR4 RAM and 1TB SSD. It is also equipped with one NVIDIA RTX-2080TI GPU for model training and inference. PostgreSQL 13.1 is installed and configured with 4GB shared buffers.

6.2 Query Performance

We compare Lero with other optimizers for the two scenarios in Section 6.1. For the hyper-parameters of Lero, we set the cardinality tuning factor $\alpha = 10$ and q-error upper bound $\Delta = 10^2$, so the set of scaling factors is $F_\alpha^\Delta = \{10^{-2}, 10^{-1}, 1, 10, 10^2\}$. For Bao and Bao+, we use the same family of 48 hint sets [30] for plan generation. As in Bao's original implementation [31], Bao and Bao+ select the same plan as PostgreSQL's optimizer for the first 100 queries while their machine learning models are trained and warmed up.

6.2.1 Performance with Stable Models. We first compare the performance of different optimizers, after Bao, Bao+, and Lero have been deployed for a while and the model training converges after seeing all training queries. We will then use the learned optimizers to process the unseen test queries. Table 1 reports their performance on finishing all test queries on all benchmarks. "Fastest Found Plan" refers to the fastest plan generated by exhaustively search for each query. Overall, Lero achieves the best performance, compared with Bao, Bao+, and PostgreSQL's optimizer. Its performance is close to the fastest found plans on STATS, TPC-H and TPC-DS.

- Lero's execution time is 70%, 44%, 21% and 13% less than PostgreSQL's native query optimizer on IMDB, STATS, TPC-H, and TPC-DS, respectively. This demonstrates Lero's practical value and advantage over the traditional optimizer in this scenario.
- Lero's execution time is 26%, 25%, 37% and 5% less than Bao on IMDB, STATS, TPC-H and TPC-DS, respectively. This verifies the effectiveness of our *learning-to-rank* paradigm and the *pairwise* trained comparator model (rather than a model predicting the exact latency). Moreover, compared with the hint set tuning approach in Bao, the plan explorer in Lero is able to generate better and more diversified plans, as described in Section 4.2, for the comparator model to learn and achieve superior query performance. Detailed comparison and experimental analysis of different plan exploration strategies will be given in Section 6.6.
- Bao+ learns from all the candidate plans generated by hint set tuning and achieves a decent performance improvement over Bao. Nevertheless, Lero's execution time is still around 20% less than Bao+ on the three benchmarks, due to Lero's more effective pairwise learning-to-rank model and plan exploration strategy.

Query Optimizer	Plans in $\mathcal{P}_1$			Plans in $\mathcal{P}_5$		
	STATS	IMDB	TPC-H	STATS	IMDB	TPC-H
PostgreSQL	0.137	0.000	0.167	0.459	0.336	0.867
Bao	0.144	0.159	0.000	0.226	0.230	0.167
Bao+	0.096	0.221	**0.833**	0.171	0.319	**1.000**
Lero	**0.425**	**0.442**	0.733	**0.925**	**0.708**	**1.000**

Table 2: Rank-specific metrics for different query optimizers.

6.2.2 Analysis of Performance Improvement/Regression. Figure 5 compares the per-query execution time of each learned optimizer (Bao, Bao+ and Lero) with PostgreSQL on the 146 test queries of STATS. We do not plot the 47 queries, for each of which PostgreSQL and all the three learned optimizers choose the same plan for execution. We sort all remaining queries by latency differences between each optimizer and PostgreSQL from slowdown to speedup to visualize performance regression/improvement of Lero and others.

In comparison to Bao and Bao+, Lero significantly reduce performance regressions and brings much more performance gains. Only 9 queries (6.2%) are slowed down in Lero (for more than one second) among 146 queries, while 33 queries are accelerated significantly. Bao and Bao+ cause regressions for 29 queries each, which are even more than the number of queries they are able to improve (26 in Bao and 24 in Bao+). When performance regression does happen in Lero, the relative slowdown of execution time, however, is much smaller. For instance, for the STATS workload, the maximum slowdown in Lero is 246s, which translates to a relative performance regression of 5.2%, while the maximum slowdown in Bao is 1,276s with a relative performance regression of 46.5%.

6.2.3 Rank-Specific Analysis. For a query optimizer, what we care about is whether the selected plan P^* is truly the best one; if not, whether it is among the top-k best plans. To this end, let $\mathcal{P}_k$ be the set of top-k fastest plans generated by exhaustively search for each query (in terms of execution time). For different benchmarks, we record the ratio of queries for which P^* falls into $\mathcal{P}_k$. Table 2 lists results for each query optimizer for $\mathcal{P}_1$ and $\mathcal{P}_5$. We find that more than 40% of plans found by Lero is the exact best plan, and more than 70% of plans is in the top-5 best plans. The percentage of plans selected by Lero falling into $\mathcal{P}_1$ and $\mathcal{P}_5$ is much higher than others.

6.2.4 Performance Curves since Deployment. We meter the performance of learned optimizers since their deployment for each workload. In Figure 6, we report their performance curves, i.e., accumulated execution time of the best plans chosen by different optimizers in each workload (see Section 6.1 for the definition). The "Fastest Found Plan" curve refers to the conceptual latency lower bound as is defined in Section 6.2.1.

- The training of Lero converges faster. After seeing 200 queries in IMDB, 100 queries in STATS, TPC-H and TPC-DS respectively, Lero's performance is consistently better than PostgreSQL and other learned optimizers. While Bao and Bao+ eventually outperform PostgreSQL, they do so after seeing many more queries. This confirms that training Lero's learning-to-rank model for query optimization is much more effective than training the latency prediction models in Bao and Bao+.

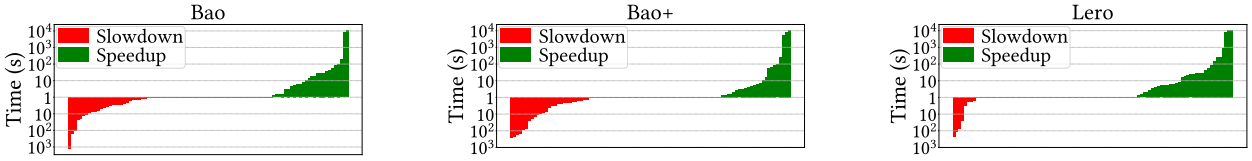


Figure 5: Per-query execution time of different query optimizers in comparison with PostgreSQL on the STATS benchmark.

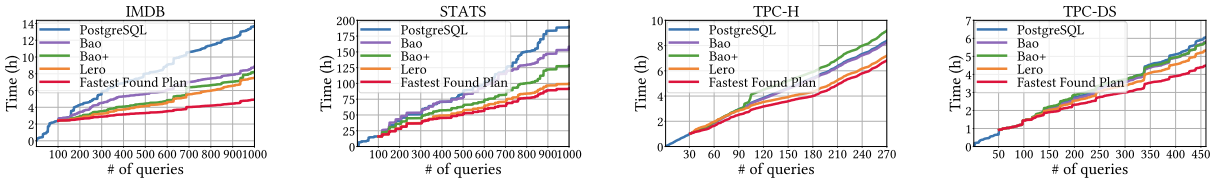


Figure 6: Performance curve of different query optimizers since deployment on benchmarks.

Time (in millisecond)	PostgreSQL	Bao/Bao+	Lero
IMDB	842	856	1,736
STATS	7	8.1	16
TPC-H	5.3	6.2	12.6
TPC-DS	6.7	8.5	15.8

Table 3: Average query optimization time per query.

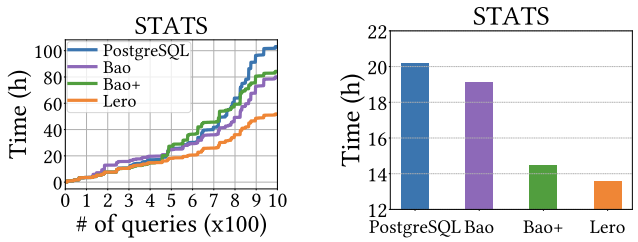
- The performance gaps between Lero and Bao/Bao+ gradually enlarge as more and more queries are executed. Eventually, Lero brings much more significant performance gain over PostgreSQL than Bao and Bao+ do. Its total execution time (right-end of each in Figure 6) is, e.g., 47%, 38% and 23% less than PostgreSQL, Bao and Bao+, respectively, on STATS.
- Lero's performance is more robust. It consistently outperforms PostgreSQL on all the benchmarks while the performance of Bao and Bao+ is sometimes worse than PostgreSQL (e.g. on TPC-H and TPC-DS). This is due to the intrinsic hardness of latency prediction which could require more training data and more time to converge in order to demonstrate any performance gain.

6.3 Query Optimization Cost

While achieving significant improvement in query execution performance, Lero spends extra query optimization time in generating a list of candidate plans for an input query and applying the comparator model to pick the best candidate. The average query optimization time per-query of different optimizers is reported in Table 3. We observe that this extra cost in query optimization is very low. In particular, the total extra cost is only at most 2.4% of the total query execution time on IMDB and less than 0.1% on STATS, TPC-H and TPC-DS. Whereas, Lero saves 13% to 70% execution time in comparison with PostgreSQL on these benchmarks.

6.4 Adapting to Dynamic Data

We now examine the performance of learned query optimizers on dynamic data. We use the STATS dataset for the experiment as each tuple is associated with a time stamp. We split the data by time. Initially, the earliest 50% of the data is stored in the database. After the first 200 queries are executed on this database, we add 12.5% of the data into the database in the order of time stamps every 200 following queries. The goal is to evaluate how well different learned optimizers adapt to dynamic data by updating their models..



(a) Performance since deployment (b) Performance with stable models

Figure 7: Performance of optimizers on dynamic data.

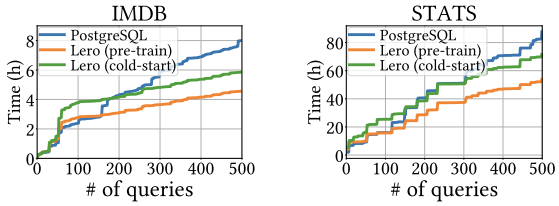


Figure 8: Effects of pre-training on Lero's performance curve.

We report the *performance curve since deployment* for each optimizer on dynamic data in Figure 7(a), as well as their *performance with stable models* in Figure 7(b).

Lero outperforms PostgreSQL, Bao, and Bao+ in both settings. After 1000 queries, Lero's accumulated query execution time is 50% less than PostgreSQL and 38% less than Bao/Bao+ (right-end of Figure 7(a)). In parallel to query processing, all the learned optimizers continuously refine their models to adapt to the dynamic changes of data. It turns out that Lero adapts to such data changes better than the other two learned optimizers. Eventually, after all the data changes are done, Lero's query performance (Figure 7(b)) is 33%, 29% and 10% better than PostgreSQL, Bao and Bao+, respectively.

In Figure 7(a), we observe Lero's robust performance, for the reason analyzed at the end of Section 3.2. Only Lero consistently performs better than PostgreSQL, while Bao is worse than PostgreSQL at the beginning and Bao+ performs worse in the middle.

6.5 Importance of Pre-Training

Lero relies on the pre-training procedure to learn from the native query optimizer and better bootstrap its own model. The pre-training time is only around 5 minutes on these benchmarks. It converges fast as the native cost models often consists of functions with simple structures, i.e., piecewise linear or quadratic function

Dataset	Strategy	Average # of Plans	Plans Faster than PostgreSQL	Plans Slower than PostgreSQL
IMDB	Lero's plan explorer	9	47%	42%
	Hint set tuning	16	36%	57%
STATS	Lero's plan explorer	5	31%	49%
	Hint set tuning	19	25%	69%
TPC-H	Lero's plan explorer	7	24%	64%
	Hint set tuning	15	6%	88%

Table 4: Number and quality of unique candidate plans generated by different plan exploration strategies.

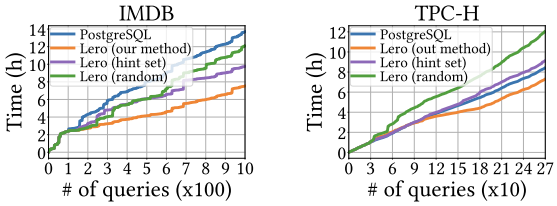


Figure 9: Performance curves of Lero since deployment with different plan exploration strategies.

of estimated cardinalities of sub-queries with magic constants as co-coefficients for different operators.

In this experiment, we evaluate the impact of the pre-training procedure on Lero's performance. We start with either a pre-trained comparator model, or a cold-start model with random parameters; both are continuously trained/updated as more and more queries are executed. Figure 8 illustrates their performance curves on IMDB and STATS for the first 500 queries (the results on TPC-H are similar, so the figure is omitted due to the space constraint).

With pre-training, for the initial 100 queries, Lero could generate almost the same plans as PostgreSQL since its comparator is pre-trained to fit PostgreSQL's native cost model; thus, their performances are very close (this can be also observed in our former experiments where Lero is pre-trained). Note that Lero does not need to execute any query during the pre-training procedure, as introduced in Section 3.2. Without pre-training, Lero's performance can be worse than PostgreSQL for the first 200 queries.before it eventually catches up. After executing nearly 300 queries, the slope of the two methods are almost the same. This indicates that Lero with a cold-start could catch up but with more training time.

Proper pre-training from the knowledge of the native query optimizer gives Lero a good starting point and also accelerates its model's convergence. With pre-training, Lero could consistently outperform PostgreSQL after the initial 100 queries, while with a cold-start comparator, Lero consistently outperforms PostgreSQL only after seeing nearly 200 queries.

6.6 Plan Exploration Strategies

Recall that, for each query, the plan explorer in Lero generates a list of candidate plans, which need to include some truly good plans (for query optimization) and be diversified (for model to learn new knowledge). Besides the plan exploration strategy described in Figure 3, we also implement two other alternative approaches in Lero to compare their performance: 1) the hint set-based strategy introduced in Bao and 2) a random strategy which randomly generates a number of plans.

Table 4 compares the quality and diversity of candidate plans generated by our plan explorer with those generated by the hint set-based strategy. On the 3rd column, we report the number of unique plans without duplication. On one hand, even with better performance, Lero's plan explorer actually generates fewer candidate plans per query on average than the hint set-based strategy (duplicate plans may be output in different iterations of lines 4-8 in Figure 3), which helps bring down the exploration cost. On the other hand, a higher percentage of candidate plans generated by Lero's plan explorer run faster than the plan generated by PostgreSQL's native optimizer, which proves such candidate plans worthy to explore and learn further. With cardinality tuning, Lero's plan explorer is able to generate more diversified plans while the hint set-based strategy typically generates plans with minor difference.

Figure 9 reports performance curves of Lero using different plan exploration strategies on IMDB and TPC-H (performance on STATS is similar to the one on IMDB, and thus is omitted due to the space limits). Both Lero's plan explorer and the hint set-based tuning strategy perform much better than the random strategy. We also observe that replacing Lero's plan explorer with the hint set-based tuning strategy performs worse. The hint set-based strategy has some intrinsic limitations, as analyzed at the beginning of Section 4. Lero's plan explorer generates good and diversified plans for the pairwise comparator model to explore and learn more effectively.

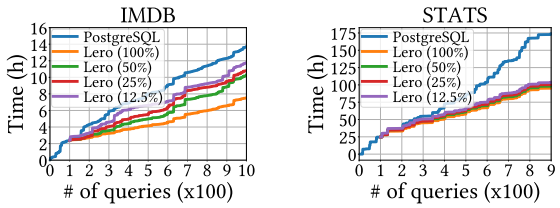


Figure 10: Performance curves of Lero since deployment with different amounts of idle resource.

6.7 Effects of Idle Resource

We examine the performance curve of Lero since deployment with different amounts of idle computation resource. By default, when an incoming query Q_{t+1} is to be processed, the comparator model in Lero is trained and updated using execution statistics of different candidate plans for earlier queries $Q_1, Q_2, \ldots, Q_t$. Such statistics are collected by running candidate plans in idle resource. When the idle resource is limited, we may only be able to finish executing candidate plans for a fraction of earlier queries $Q_1, Q_2, \ldots, Q_t$. With only the first 50%, 25%, and 12.5% earlier queries having all of their candidates executed on idle resource, we re-run the experiments in Section 6.2.4. We demonstrate in Figure 10 that the performance of Lero can still be improved over time and converge eventually.

Even with limited idle resource, the initial performance of Lero is no worse than PostgreSQL, thanks to the pre-training procedure. However, at the beginning, the quality of the selected plans is indeed worse than the default setting (when 100% of earlier queries have their candidate plans executed on idle resource), because the comparator model is not fully trained. We observe that the model still converges fast even with very limited idle resource. For instance, when only 12.5% queries have their candidate plans executed and explored, the model converges after the first 500 and 200 queries on IMDB and STATS, respectively.

Speedup Ratio	Random Split		Slow Split	
	Train	Test	Train	Test
Bao	1.62×	1.49×	1.17×	1.05×
Bao+	2.93×	1.58×	4.01×	1.74×
Balsa	2.46×	1.55×	1.23×	2.31×
Lero	3.54×	1.59×	4.38×	1.34×

Table 5: Learned optimizers on IMDB with original JOB.

Regardless of under which configuration, Lero becomes stable eventually and performs almost the same (curves become parallel towards the right end). This implies that limited idle resource would definitely slow down the model's learning and convergence, but has little impact on the ultimate performance of Lero.

6.8 Comparison with Balsa

We compare Lero with Balsa on the IMDB dataset with the original JOB workload. We use its open-source implementation in [1] and prepare the dataset in the same way as [51]. Specifically, the 113 queries in JOB are split into training set (with 94 queries) and testing set (with 19 queries) in two ways: 1) random split and 2) the test set consists of the 19 slowest-running queries when planned by an expert optimizer. We run Balsa using its default settings and compare its plan quality with our Lero, as well as Bao and Bao+.

Table 5 exhibits the speedup ratio of each learned query optimizer in comparison to PostgreSQL's native optimizer. The performance of Balsa matches or outperforms Bao, which is consistent with the results in [51]. Both Bao and Balsa try to predict plan latency to guide plan search, while Balsa has higher freedom of exploration (Bao's search space is limited by hint set tuning) [51]. Thus, it performs better than Bao on small and stable workload. However, even after Balsa's model is comprehensively trained on this small JOB workload, it still performs worse than Bao+ and Lero.

7 RELATED WORK

Learning to Optimize Queries. Recently, there is a flurry of research to apply machine learning techniques to various components in query optimization [59]. The majority of them focus on learned cardinality estimation, using either query-driven or data-driven approaches. Query-driven methods [12, 22, 27] apply learned models to map featurized query to its cardinality. Data-driven methods [18, 45, 50, 52, 53, 60] use different generation models to directly learn the underlying data distribution. Some benchmark evaluations [16, 48] have examined their superiority and limitations. Other works focus on refining traditional cost models and plan enumeration algorithms. For learned cost models, [41] and [55, 57] utilize TreeLSTM and convolution models to learn cost of single and concurrent queries, respectively. Plan enumeration is often modelled as a reinforcement learning problem on deciding the best join order of tables. [17, 23] and [54] use simple neural networks and TreeLSTM model as value networks for join order selection, respectively. [44] further considers how to adjust the join order on the fly. These works only optimize an individual component in query optimizer, which does not necessarily improve the overall performance.

Besides them, recent works [33, 51] provide end-to-end learned solutions for query optimization, and [32] learns to steer a native query optimizer using hint set tuning. However, as analyzed in Section 1.1, they suffer from lots of deficiencies arising from predicting the cost or latency. Based on plan exploration and pairwise plan comparison, Lero learns the difference between plan pairs and learns to improve the end-to-end quality of query optimization.

Learning-to-Rank Paradigm. Lero follows a learning-to-rank paradigm, which is a class of learning techniques to train models for ranking tasks [21, 28, 47]. It has been widely applied for, e.g., document retrieval, collaborative filtering, and recommendation systems. Based on how ranking is generated, the learning-to-rank techniques could be classified into pointwise [14, 26], pairwise [13, 29] and listwise [38, 42] approaches. Among them, pairwise approach learns a classifier on a pair of items to identify which one is better. To our best knowledge, we are the first to apply a *pairwise learning-to-rank* paradigm to develop a learned query optimizer.

Learning to Tune Indexes. The task of indexing tuning is to find the set of indexes that fits in a given storage budget and results in the lowest execution cost for a given workload of queries. A traditional index tuner [3, 5, 46] first searches for the optimal index configuration for each query (query-level search), and then enumerates different sets of those index configurations to search for the optimal index set for the workload under the storage budget (workload-level search). Both phases need a key API that compares the execution costs of two plans of the same query corresponding to different index configurations. Traditional index tuners rely on optimizer's estimates for such comparisons, while [10] trains a binary classifier to this end resulting in significantly higher accuracy.

While the query-level search in Ding et al.'s work [10] can be regarded as an application of learning-to-rank in index tuning, it has the following two fundamental differences from our Lero.

First, the index tuner implemented in [10] is similar to state-of-the-art tuners [3, 5, 46], except that the trained classifier, as a pairwise comparator model, is invoked during query/workload-level search as an API to determine whether the plan of a query under a new index configuration is improved in comparison to the one under the initial configuration. For a completely different task, query optimization, Lero is equipped with two carefully designed components, plan explorer and plan comparator model. These two components are coupled closely in our pairwise learning-to-rank paradigm. They work together to explore the plan space for reasonable candidate plans and, meanwhile, collect valuable execution statistics to improve the comparator model continuously.

Second, the comparator models in [10] and Lero both compare the performance of two plans; however, with different goals in the two tasks, the ways how features are encoded and the model architectures in the two comparators are different. In [10], since the goal is index selection, what really matters is the amount of work done in the plan by the each type of operators, with or without indexes. Thus, for each *type* of operators in a plan, two features can be created: i) the total estimated costs and ii) the total (in a weighted way) estimated amounts of data processed by them. In this way, a plan is flattened as a vector, and the classifier is trained on pairs of vectors (with a label indicating which one is better in a pair) for the comparison purpose. In Lero, the goal is plan selection, and thus we need finer-grained information on each node (e.g., operator type, estimates, tables involved) unfolded and use the tree convolution operation to slide multiple triangle shaped filters over each node

to capture more structural information and correlation between tables. After such learnable embedding layers for two plans, another learnable comparison layer decides which one is better.

8 CONCLUSIONS

We propose Lero, a *learning-to-rank* query optimizer. First, Lero applies learning-to-rank machine learning techniques to query optimization. We argue that it is an overkill to develop machine models to predict the exact execution latency or cardinalities in terms of query optimization. Instead, Lero adopts a *pairwise* approach to train a binary classifier to compare any two plans, which is proven to be more efficient and effective. Second, Lero takes advantage of decades of wisdom of database research and jointly works with the native query optimizer to improve optimization quality. Third, Lero is equipped with a plan exploration strategy, which enables Lero to explore new optimization opportunities more effectively. Finally, an extensive evaluation on our implementation on top of PostgreSQL demonstrates Lero's superiority in query performance and its ability to adapt to changing data and workload.

With its non-intrusive design, Lero can be implemented on top of any existing DBMSs. We are in the process of implementing Lero on top of a commercial DBMS. Our preliminary results also confirm Lero's significant improvements over its native query optimizer.

REFERENCES

[1] 2022. Balsa Implementation. https://github.com/balsa-project/balsa.

[2] 2022. Lero on PostgreSQL. https://github.com/AlibabaIncubator/Lero-on-PostgreSQL.

[3] Sanjay Agrawal, Surajit Chaudhuri, Lubor Kollár, Arunprasad P. Marathe, Vivek R. Narasayya, and Manoj Syamala. 2004. Database Tuning Advisor for Microsoft SQL Server 2005. In *VLDB*. 1110–1121.

[4] Ronnie Chaiken, Bob Jenkins, Per-Åke Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. 2008. Scope: easy and efficient parallel processing of massive data sets. *PVLDB* 1, 2 (2008), 1265–1276.

[5] Surajit Chaudhuri and Vivek R. Narasayya. 1997. An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server. In *VLDB*. 146–155.

[6] Jianmin Chen, Xinghao Pan, Rajat Monga, Samy Bengio, and Rafal Jozefowicz. 2016. Revisiting distributed synchronous SGD. *arXiv:1604.00981* (2016).

[7] Transaction Processing Performance Council(TPC). 2021. TPC-DS Vesion 2 and Version 3. http://www.tpc.org/tpcds/.

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

[9] Atreyee Dey, Sourjya Bhaumik, Harish Doraiswamy, and Jayant R. Haritsa. 2008. Efficiently approximating query optimizer plan diagrams. *Proc. VLDB Endow.* 1, 2 (2008), 1325–1336.

[10] 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 *SIGMOD*. 1241–1258.

[11] Harish Doraiswamy, Pooja N. Darera, and Jayant R. Haritsa. 2008. Identifying robust plans through plan diagram reduction. *Proc. VLDB Endow.* 1, 1 (2008), 1124–1140.

[12] Anshuman Dutt, Chi Wang, Azade Nazi, Srikanth Kandula, Vivek Narasayya, and Surajit Chaudhuri. 2019. Selectivity estimation for range predicates using lightweight models. *PVLDB* 12, 9 (2019), 1044–1057.

[13] Yoav Freund, Raj Iyer, Robert E Schapire, and Yoram Singer. 2003. An efficient boosting algorithm for combining preferences. *Journal of machine learning research* 4, Nov (2003), 933–969.

[14] Norbert Fuhr. 1989. Optimum polynomial retrieval functions based on the probability ranking principle. *ACM Transactions on Information Systems* 7, 3 (1989), 183–204.

[15] Yuxing Han. 2021. Github repository: STATS End-to-End CardEst Benchmark. https://github.com/Nathaniel-Han/End-to-End-CardEst-Benchmark.

[16] Yuxing Han, Ziniu Wu, Peizhi Wu, Rong Zhu, Jingyi Yang, Tan Wei Liang, Kai Zeng, Gao Cong, Yanzhao Qin, Andreas Pfadler, Zhengping Qian, Jingren Zhou, Jiangneng Li, and Bin Cui. 2021. Cardinality Estimation in DBMS: A Comprehensive Benchmark Evaluation. *PVLDB* 15, 4 (2021), 752–765.

[17] Todd Hester, Matej Vecerik, Olivier Pietquin, Marc Lanctot, Tom Schaul, Bilal Piot, Dan Horgan, John Quan, Andrew Sendonaris, Ian Osband, et al. 2018. Deep q-learning from demonstrations. In *AAAI*. 3223–3230.

[18] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. 2020. DeepDB: learn from data, not from queries! *PVLDB* 13, 7, 992–1005.

[19] Fürnkranz Johannes and Hüllermeier Eyke. 2011. *Preference Learning*. Preference Learning.

[20] John J Kanet and V Sridharan. 2000. Scheduling with inserted idle time: problem taxonomy and literature review. *Operations Research* 48, 1 (2000), 99–110.

[21] Alexandros Karatzoglou, Linas Baltrunas, and Yue Shi. 2013. Learning to rank for recommender systems. In *RecSys*. ACM, 493–494.

[22] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter Boncz, and Alfons Kemper. 2019. Learned cardinalities: Estimating correlated joins with deep learning. In *CIDR*.

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

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

[25] Jiexing Li, Arnd Christian König, Vivek Narasayya, and Surajit Chaudhuri. 2012. Robust estimation of resource consumption for sql queries using statistical techniques. *arXiv:1208.0278* (2012).

[26] Ping Li, Qiang Wu, and Christopher Burges. 2007. Mcrank: Learning to rank using multiple classification and gradient boosting. *Advances in neural information processing systems* 20 (2007).

[27] Jie Liu, Wenqian Dong, Qingqing Zhou, and Dong Li. 2021. Fauce: fast and accurate deep ensembles with uncertainty for cardinality estimation. *PVLDB* 14, 11 (2021), 1950–1963.

[28] Tie-Yan Liu. 2009. Learning to Rank for Information Retrieval. *Foundations and Trends in Information Retrieval* 3, 3 (2009), 225–331.

[29] Weiwen Liu, Qing Liu, Ruiming Tang, Junyang Chen, Xiuqiang He, and Pheng Ann Heng. 2020. Personalized Re-ranking with Item Relationships for E-commerce. In *CIKM*. 925–934.

[30] Ryan Marcus. 2020. Bao appendix. https://rmarcus.info/appendix.html.

[31] Ryan Marcus. 2020. Github repository: Bao for PostgreSQL. https://github.com/learnedsystems/BaoForPostgreSQL.

[32] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2021. Bao: Making learned query optimization practical. In *SIGMOD*. 1275–1288.

[33] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: A Learned Query Optimizer. *PVLDB* 12, 11 (2019), 1705–1718.

[34] Ryan Marcus and Olga Papaemmanouil. 2019. Plan-structured deep neural network models for query performance prediction. *PVLDB* 12, 11 (2019), 1733–1746.

[35] Guido Moerkotte, Thomas Neumann, and Gabriele Steidl. 2009. Preventing Bad Plans by Bounding the Impact of Cardinality Estimation Errors. *VLDB* 2, 1 (2009), 982–993.

[36] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional neural networks over tree structures for programming language processing. In *AAAI*. 1287–1293.

[37] 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 *SIGMOD*. 2557–2569.

[38] Liang Pang, Jun Xu, Qingyao Ai, Yanyan Lan, Xueqi Cheng, and Jirong Wen. 2020. Setrank: Learning a permutation-invariant ranking model for information retrieval. In *SIGIR*. 499–508.

[39] Naveen Reddy and Jayant R. Haritsa. 2005. Analyzing Plan Diagrams of Database Query Optimizers. In *VLDB*. 1228–1240.

[40] P Griffiths Selinger, Morton M Astrahan, Donald D Chamberlin, Raymond A Lorie, and Thomas G Price. 1989. Access path selection in a relational database management system. In *Readings in Artificial Intelligence and Databases*. Elsevier, 511–522.

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

[42] Robin Swezey, Aditya Grover, Bruno Charron, and Stefano Ermon. 2021. Pirank: Scalable learning to rank via differentiable sorting. *Advances in Neural Information Processing Systems* 34 (2021), 21644–21654.

[43] William R Thompson. 1933. On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika* 25, 3-4 (1933), 285–294.

[44] Immanuel Trummer, Junxiong Wang, Deepak Maram, Samuel Moseley, Saehan Jo, and Joseph Antonakakis. 2019. Skinnerdb: Regret-bounded query evaluation via reinforcement learning. In *SIGMOD*. 1153–1170.

[45] Kostas Tzoumas, Amol Deshpande, and Christian S Jensen. 2011. Lightweight graphical models for selectivity estimation without independence assumptions. *PVLDB* 4, 11 (2011), 852–863.

[46] Gary Valentin, Michael Zuliani, Daniel C. Zilio, Guy M. Lohman, and Alan Skelley. 2000. DB2 Advisor: An Optimizer Smart Enough to Recommend Its Own Indexes. In *ICDE*. 101–110.

[47] Saúl Vargas and Pablo Castells. 2011. Rank and relevance in novelty and diversity metrics for recommender systems. In *Proceedings of the fifth ACM conference on Recommender systems*. 109–116.

[48] Xiaoying Wang, Changbo Qu, Weiyuan Wu, Jiannan Wang, and Qingqing Zhou. 2021. Are We Ready For Learned Cardinality Estimation? *PVLDB* 14, 9 (2021), 1640–1654.

[49] Wentao Wu, Yun Chi, Shenghuo Zhu, Junichi Tatemura, Hakan Hacigümüs, and Jeffrey F Naughton. 2013. Predicting query execution time: Are optimizer cost models really unusable?. In *ICDE*. 1081–1092.

[50] Ziniu Wu and Amir Shaikhha. 2020. BayesCard: A Unified Bayesian Framework for Cardinality Estimation. *arXiv:2012.14743* (2020).

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

[52] Zongheng Yang, Amog Kamsetty, Sifei Luan, Eric Liang, Yan Duan, Xi Chen, and Ion Stoica. 2021. NeuroCard: One Cardinality Estimator for All Tables. *PVLDB* 14, 1 (2021), 61–73.

[53] Zongheng Yang, Eric Liang, Amog Kamsetty, Chenggang Wu, Yan Duan, Xi Chen, Pieter Abbeel, Joseph M Hellerstein, Sanjay Krishnan, and Ion Stoica. 2019. Deep unsupervised cardinality estimation. *PVLDB* 13, 3 (2019), 279–292.

[54] Xiang Yu, Guoliang Li, Chengliang Chai, and Nan Tang. 2020. Reinforcement learning with tree-lstm for join order selection. In *ICDE*. 1297–1308.

[55] Johan Kok Zhi Kang, Sien Yi Tan, Feng Cheng, Shixuan Sun, and Bingsheng He. 2021. Efficient Deep Learning Pipelines for Accurate Cost Estimations Over Large Scale Query Workload. In *SIGMOD*. 1014–1022.

[56] Li Zhou. 2015. A survey on contextual multi-armed bandits. *arXiv:1508.03326* (2015).

[57] Xuanhe Zhou, Ji Sun, Guoliang Li, and Jianhua Feng. 2020. Query performance prediction for concurrent queries using graph embedding. *PVLDB* 13, 9 (2020), 1416–1428.

[58] Rong Zhu, Wei Chen, Bolin Ding, Xingguang Chen, Andreas Pfadler, Ziniu Wu, and Jingren Zhou. 2022. Lero: A Learning-to-Rank Query Optimizer [Full Version]. *https://github.com/AlibabaIncubator/Learned-Query-Optimizer-by-Ranking-Plans/blob/main/lerofull.pdf* (2022).

[59] Rong Zhu, Ziniu Wu, Chengliang Chai, Andreas Pfadler, Bolin Ding, Guoliang Li, and Jingren Zhou. 2022. Learned Query Optimizer: At the Forefront of AI-Driven Databases.. In *EDBT*. 1–4.

[60] Rong Zhu, Ziniu Wu, Yuxing Han, Kai Zeng, Andreas Pfadler, Zhengping Qian, Jingren Zhou, and Bin Cui. 2021. FLAT: Fast, Lightweight and Accurate Method for Cardinality Estimation. *PVLDB* 14, 9 (2021), 1489–1502.

APPENDIX

A COMPARATOR WITH *d-dim* EMBEDDING

Our plan comparator model CmpPlan (described in Section 3) could be easily extend from 1-*dim* plan embedding to use *d-dim* ($d > 1$) plan embedding. We use the same way on encoding plans and slightly modify the plan comparator model (shown in Figure 2) as follows:

(1) In the plan embedding layers, we change the output dimension of last embedding layer to d so that the plan embedding $\mathsf{PlanEmb}(P)$ becomes a *d-dim* vector for any plan P.
(2) We change the comparison layer to a learnable linear layer which takes two *d-dim* vectors $\mathsf{PlanEmb}(P_1)$ and $\mathsf{PlanEmb}(P_2)$ as inputs, and outputs 0 (P_1 is better) or 1 (P_2 is better).

Later, we discuss how to select the best plan under *d-dim* embedding (in A.1) and the performance of plan comparator w.r.t. the dimensions of plan embedding (in A.2).

A.1 Selecting the Best Plan using Comparators

By Section 2, the plan comparator $\mathsf{CmpPlan}(P_1, P_2)$ is an oracle comparing any two plans P_1 and P_2 of a query. Given a list of candidate plans $P_1, P_2, \ldots, P_n$ of a query, let

$$\mathrm{Wins}(P_i) = |\{P_j \mid \mathsf{CmpPlan}(P_i, P_j) = 0, j \neq i\}|, \tag{7}$$

be the number of plans whose latencies are worse than P_i. $\mathrm{Wins}(P_i)$ defines a full order of all candidate plans. Obviously, the best candidate plan P^* has $\mathrm{Wins}(P^*) = n - 1$.

In Section 3, our comparator model is to learn the above oracle. We also use CmpPlan to denote the learned comparator model. After a comparator is fully trained, we can use it to pick the best plan among candidates $P_1, \ldots, P_n$ of a possibly unseen query Q. In the general design where we have $d \geq 1$ for the plan embedding, we do not require a fully trained CmpPlan to preserve transitivity, i.e., P_1 is better than P_2 and P_2 is better than P_3 imply that P_1 is better than P_3. In order to pick the best plan among candidates $P_1, \ldots, P_n$, we can invoke CmpPlan $n(n-1)$ times to compare all pairs of candidates. In general the output $\mathsf{CmpPlan}(P_i, P_j) \in (0, 1)$ gives a soft prediction (i.e., the predicted probability of whether P_i or P_j is better). We use the algorithm in Eq. (3) to pick the better one, and define *randomized wins* as

$$\mathrm{RandomWins}(P_i) = |\{P_j \mid \mathcal{A}(P_i, P_j) = P_i, j \neq i\}|.$$

Accordingly, we choose the best plan as

$$P^* = \underset{P_i \in \{P_1, \ldots, P_n\}}{\arg\max} \ \mathrm{RandomWins}(P_i)$$

with the most randomized wins–ties are broken arbitrarily.

In the more practical design of CmpPlan with 1-*dim* plan embedding (Section 3), we do not need to derive $\mathrm{RandomWins}(P_i)$ as above. Instead, the output of the sub-model PlanEmb defines a *total order* on all candidate plans. For any pair of plans (P_i, P_j), if $\mathsf{PlanEmb}(P_i) > \mathsf{PlanEmb}(P_j)$, we always have $\mathsf{CmpPlan}(P_1, P_2) > 0.5$, i.e., P_j is more preferable than P_i with higher probability than vice versa. Therefore, we could select $P^* = \arg\min_{P_i} \mathsf{PlanEmb}(P_i)$ with the lowest value of PlanEmb as the best plan for execution.

We can show that, P^* chosen in this way is indeed the one with the most randomized wins *in expectation* (thus, the choice is equivalent to the one for the general design, in expectation).

Proposition 3. *$P^* = \arg\min_{P_i} \mathsf{PlanEmb}(P_i)$ chosen with the lowest 1D embedding also wins the most: it satisfies*

$$P^* = \underset{P_i \in \{P_1, \ldots, P_n\}}{\arg\max} \ \mathbb{E}[\mathrm{RandomWins}(P_i)] .$$

Proof. By the model construction, for each P_i, we have

$$\begin{aligned}
\mathbb{E}[\mathrm{RandomWins}(P_i)] &= \sum_{j \in [n], j \neq i} \Pr\big[\mathcal{A}(P_i, P_j) = P_i\big] \\
&= \sum_{j \in [n], j \neq i} (1 - \mathsf{CmpPlan}(P_i, P_j)) = \sum_{j \in [n], j \neq i} \mathsf{CmpPlan}(P_j, P_i) \\
&= \left(\sum_{j \in [n]} \phi(\mathsf{PlanEmb}(P_j) - \mathsf{PlanEmb}(P_i)) \right) - \phi(0).
\end{aligned} \tag{8}$$

From Eq. (8), it is straightforward that $P^* \in \{P_1, \ldots, P_n\}$ with the lowest $\mathsf{PlanEmb}(\cdot)$ would maximize $\mathbb{E}[\mathrm{RandomWins}(\cdot)]$. □

A.2 Performance of Comparator with Different Dimension of Plan Embedding

We test the performance of Lero with different dimensions of plan embedding. We set $d = \{1, 2, 4, 8, 16\}$, train Lero on the training workload and then test its performance on the test workload. Table 6 reports the execution time of Lero with different dimensions of plan embedding on IMDB and STATS benchmarks. Similar results are also observed on TPC-H and TPC-DS benchmarks.

Time (in hour)	IMDB	STATS
PostgreSQL	1.15	20.19
Lero ($d = 1$)	0.35	11.32
Lero ($d = 2$)	0.77	20.74
Lero ($d = 4$)	0.69	21.2
Lero ($d = 8$)	0.49	21.38
Lero ($d = 16$)	0.59	21.19

Table 6: Performance of Lero with different dimensions of plan embedding.

Interestingly, we observe that the performance of Lero does not necessarily increase with higher dimensional embeddings. On the contrary, Lero with complex embedding ($d > 1$) has much worse performance than using the simplest 1-*dim* plan embedding. Specifically, on IMDB, the the execution time of Lero with a larger $d \in \{2, 4, 8, 16\}$ is 1.4 times to 2.2 times as long as the case with $d = 1$. On STATS, the execution time of Lero with $d > 1$ is around 1.8 times longer than $d = 1$, even longer than the execution time of the native PostgreSQL. The reason could be that, with $d > 1$, the embedding model PlanEmb tries to summarize more sophisticated information in plan embeddings, but also requires larger training

dataset and more training time to obtain more accurate embeddings. Therefore, the training workload may be not enough to comprehensively capture enough information for plan comparison, so the selected plan may be not the best one.

In our setting, Lero using 1-*dim* plan embedding already attains significant performance improvement in comparison to the native query optimizer. More over, when $d > 1$, it is not convenient to select the best plan as $\mathsf{PlanEmb}(P)$ does not define a total order on all plans P. Thus, in our design choice, we use the simplest 1-*dim* plan embedding in Lero.

B EXPLORATION HEURISTIC BASED ON PLAN DIAGRAM

Errors in estimating predicate selectivities can be propagated from bottom to top during the plan search, and incur a wrong join order and sub-optimal choices of join types. A *plan diagram* [9, 11] can be constructed by varying values of parameters in one or more predicates of the query Q, so that the optimizer generates a set of different plans. For example, for a query with a parameterized predicate "@lower ≤ Age ≤ @upper", the optimizer may generated different plans by setting (@lower, @upper) as $(20, 30)$ or $(10, 40)$. These plans form a diagram of different regions on the 2D space (@lower, @upper): within each region, the optimizer outputs the same plan.

It is shown in [9, 11] that replacing selectivity error-sensitive plan choices with alternatives in the plan diagram provides potentially better performance. Therefore, we can use plans in the plan diagram as the list of candidates to explore the uncertainty from estimating predicates' selectivities in the native optimizer.

The drawbacks of this strategy are obvious. It is not affordable to generate a plan diagram by varying parameters on more than two tables, as the number of candidate plans would be too large [9]. If we tune parameters on two tables, the candidates may not be diversified enough to include plans with quality sufficiently higher than the plan generated by the native optimizer. In our evaluation on the STATS benchmark [16], even the best candidate in a plan diagram does not have significant performance improvement. Specifically, the average performance improvement of the best candidate is less than 3%, and the best candidates of only less than 5% of queries are faster than those generated by PostgreSQL's optimizer.