Eraser: Eliminating Performance Regression on Learned Query Optimizer

Lianggui Weng[1,#], Rong Zhu[1,#], Di Wu[1,2], Bolin Ding[1], Bolong Zheng[2], Jingren Zhou[1]
[1]Alibaba Group [2] Huazhong University of Science and Technology
[1]{lianggui.wlg, red.zr, woodybryant.wd, bolin.ding, jingren.zhou}@alibaba-inc.com, [2]bolongzheng@hust.edu.cn

ABSTRACT

Efficient query optimization is crucial for database management systems. Recently, machine learning models are applied in query optimizers to generate better plans. Despite such learned query optimizers have shown superiority in some benchmarks, unpredictable performance regressions prevent them from being truly applicable. To be more specific, while a learned query optimizer commonly outperforms the traditional query optimizer on average for a workload of queries, its performance regression seems inevitable for some queries due to model under-fitting and difficulty in generalization. In this paper, we propose a system called Eraser to resolve this problem. Eraser aims at eliminating performance regressions while still attaining considerable overall performance improvement. To this end, Eraser applies a two-stage strategy to estimate the model accuracy for each candidate plan, and helps the learned query optimizer select more reliable plans. The first stage serves as a coarse-grained filter that removes all highly risky plans with features that are seen for the first time. The second stage clusters plans in a more fine-grained manner and evaluate each cluster according to the estimation quality of plans for selecting final execution plan. Eraser could be deployed as a plugin on top of any learned query optimizer. We implement Eraser and demonstrate its superiority on PostgreSQL and Spark. In our experiments, Eraser eliminates most of the regressions while brings very little negative impact on the overall performance of learned query optimizers, no matter whether they perform better or worse than the traditional query optimizer. Meanwhile, it is adaptive to dynamic settings and generally applicable to different database systems.

1 INTRODUCTION

Query optimization plays a crucial role in database management systems. The goal of query optimizer is to find an optimal execution plan that minimizes a user-specified cost metric, e.g., the query execution time or resource usage. Traditional query optimizers rely on cost-based model that estimates the cost of execution plans based on simple statistics and experience-driven rules [34]. However, the estimated costs are often shown to have large errors [11, 15, 21, 37], due to unrealistic independence assumptions or over-simplified models, which heavily degrade the generated plan quality.

Recently, there has been an active line of works using learned optimizers to improve query optimization [13, 26, 28, 29, 42, 43, 46]. These optimizers apply (deep) models to learn from data and/or queries to generate better execution plans. The pipeline of a learned query optimizer often includes two main steps. First, it generates a number of candidate plans $\mathcal{P}_Q$ by some exploration strategy. Specifically, Neo [28] and Balsa [42] apply best-first and beam search method in the plan space. Bao [26], Lero [46] and HyperQO [43] tune different knobs of the traditional query optimizer to produce a number of different, possibly better, plans. Second, all candidate plans $P \in \mathcal{P}_Q$ are fed into a learned deep model to predict its cost $C(P)$, including pointwise regression model [26, 28, 43] and pairwise learning-to-rank model [13, 46]. The plan $P_r \in \mathcal{P}_Q$ that minimizes the predicted cost is selected to execute.

Challenges of Learned Query Optimizer. Despite the promising results of learned query optimizers have been shown in the literature [10, 26, 46], they still suffer inevitable drawbacks that prevent them from being truly applicable. Specifically, the executed plans selected by the learned query optimizer may be worse, sometimes even seriously worse, than the traditional native query optimizer. This phenomenon is called *performance regression*, and has been observed in all learned query optimizers [13, 26, 28, 29, 42, 43, 46].

The performance regressions are caused by numerous reasons, including but not limited to the low generalization ability of the prediction model on new data, and the under-fitting on training data due to insufficient training data, loss of features, noisy labels, inappropriate model training methods, bad hyper-parameters and dynamic data/workload. Due to the uncertainty nature of deep model, it is hard to predict its estimation accuracy. As a result, the performance regression seems inevitable in learned query optimizers. This is very harmful, or even unacceptable, to database systems which require high stability.

In the literature, there exists very little work on eliminating the performance regression, especially on learned query optimizer [47]. [23] and [19] propose methods to enhance the robustness in dynamic settings by updating models in proper time. They are post-processing methods but do not detect and eliminate regression before query execution. [43] tries to reduce the regression using the ensemble methods [3, 14, 20, 38, 41], but it is time costly and often falsely filters some truly good plans. As far as we know, until now, this problem has not been well solved.

Our Contributions. In this paper, we try to tackle this problem in a novel way. Notably, the benefit and risk of the learned models always come together. Our goal is not to eliminate any possible regression (which degenerates to the traditional query optimizer), but to eliminate it to a low level while still attaining considerable performance improvement. To this end, we design a system called Eraser, which could be deployed as an external plugin on top of any existing learned query optimizer. Eraser could be tuned to eliminate its performance regression while brings minimal impact on its performance benefit.

The key to eliminate the performance regression is to identify whether the predicted cost is accurate for each candidate plan. Based on this, we could filter out all unpromising candidate plans

# The first two authors contribute equally to this paper.

but reserve other ones with high learning accuracy for plan selection. However, learning the exact estimation accuracy is very challenging, which is as difficult as learning the accurate cost of each plan [11, 15, 21, 27, 37]. In Eraser, we try to simplify the learning tasks while still preserve enough knowledge for plan identification.

Specifically, Eraser adopt a two-stage strategy for plan identification. The first stage serves as a coarse-grained filter that qualitatively removes all highly risky plans. We observe that the prediction models are very likely to perform worse on plans with feature values not occurring in the training data, due to their low generalization ability. These plans are called *unexpected plans*. To detect how the model behaves w.r.t. each feature value, we design an *unexpected plan explorer* to divide the unexpected plan space into a number of subspace, each with one or more unseen feature values. Then, we generate plans in a small number of representative subspace. Based on the model evaluation results on these plans, we could classify all subspace into precise and imprecise. All candidate plans fall into the imprecise subspace would be filtered.

In the second stage, we learn a *segment model* to process the remaining plans in a more fine-grained manner. We observe that the performance of the prediction model is highly skewed, since it is under-fitting for some plans. To this end, the segment model groups plans in a number of clusters and associates each cluster with a reliability interval reflecting the quality of the estimation results. Based on the reliability interval, we design a plan selection method to balance the risk of regressions and the loss on benefits. Both the unexpected plan explorer and the segment model are lightweight. They work together to eliminate regressions caused by different reasons. They could also be easily updated in dynamic settings.

By comprehensive evaluations, we find that: When the learned query optimizer performs worse, i.e., even 1.1× to 2.9×, than the traditional query optimizer, Eraser could help to improve its performance to be comparable with the traditional query optimizer. When the learned query optimizer performs better than the traditional query optimizer, Eraser makes little difference on its performance. Eraser is adaptive to balance regression risks and improvement impacts to attain the best overall performance in both static and dynamic settings. Meanwhile, Eraser exhibits good generality to different underlying learned query optimizers in [13, 43, 46] and on different DBMSes, i.e., PostgreSQL and Spark [44].

Our main contributions are summarized as follows:

1) We propose a general framework subsuming existing learned query optimizers. Based on this, we rigorously define the performance regression elimination problem on learned query optimizer.

2) We design Eraser, a system that could be deployed on top of any learned query optimizer to eliminate its performance regression while preserving its performance benefit.

3) We conduct extensive experiments to evaluate the performance of Eraser in different settings.

Organization Section 2 provides some preliminaries including the background on (learned) query optimizer, formalization of the problem and analysis of existing solutions. Section 3 presents an overview of our Eraser system. Section 4 and Section 5 introduce the technical details on unexpected plan explorer and segment model, the two key components in our system, respectively. Section 6 reports the evaluation results. Finally, Section 7 concludes the paper and discusses future work.

2 PRELIMINARY

2.1 Background of Query Optimizer

In the traditional query optimizer, such as the one in PostgreSQL, for any input SQL query Q, all candidate plans are often enumerated using dynamic programming. Then, a basic cost model is applied for plan selection. It relies on estimated cardinality, which is often generated by simple statistical methods such as histogram or sampling, and experience-driven rules to predict the cost of each candidate plan. Let $C(P)$ and $\widehat{C}(P)$ denote the exact and estimated cost of query plan P, respectively. The cost $C(P)$ is a user-specified metric, e.g., execution time or I/O throughput, regarding the efficiency of executing P. Finally, the plan P_b with the minimum estimated cost is returned for execution.

Framework of Learned Query Optimizer. Recently, a number of learned query optimizers [13, 26, 28, 42, 43, 46] are proposed to provide instance-level query optimization. Their procedures could be generalized into a unified framework with two main steps. For the input query Q, a learned query optimizer first generates a set of candidate plans $\mathcal{P}_Q = \{P_0, P_1, \ldots, P_k\}$ using some plan exploration strategies. Then, a learned risk model M_r, i.e., a complex ML-based model, is applied for plan selection. M_r could predict the goodness of each plan in $\mathcal{P}_Q$ in terms of $C(P)$. The best plan $P_r \in \mathcal{P}_Q$ minimizing $\widehat{C}(P)$ is selected for execution.

We note that, different learned query optimizers apply different plan exploration strategies and risk models, but they could all be subsumed under this framework as follows:

- **Neo** [28] and **Balsa** [42] generate the candidate plans $\mathcal{P}_Q$ from scratch by best-first and beam search strategy, respectively. Then, their risk models apply the tree convolution network [29] to predict the execution time of each plan $P \in \mathcal{P}_Q$.
- **Bao** [26] steers the native traditional query optimizer with different hints to enable or prohibit certain physical operators to generate different candidate plans. Its risk model is also based on a tree convolution network with simpler features.
- **HyperQO** [43] uses different leading hints to control the join order of tables to collect candidate plans. Its risk model relies on the multi-head LSTM structure for predicting execution time.
- **Lero** [46] applies the estimated cardinality as the tuning knob for generating candidate plans. It scales the estimated cardinality by some factors to produce different (possibly better) plans. Unlike other works, Lero trains a pairwise classification model. For each pair of plans $P, P' \in \mathcal{P}_Q$, the model learns to predict which plan is better in terms of cost. The plan surpassing the most number of other plans is then selected to execute.
- **PerfGuard** [13] could support any method to generate candidate plans. It also applies the pairwise model, which incorporates graph convolution networks, for plan selection.
- Some other works replace the cardinality estimation [12, 17, 30, 48] or cost model [27, 35] in query optimizer to learned models.

They apply the same plan exploration method as the traditional query optimizer but could predict more accurate cost.

2.2 Problem Statement

The plan P_r selected by the above learned query optimizer is often shown to have a better performance than P_b. However, it may also suffer a heavy performance regression on some queries due to numerous reasons:

- the candidate set $\mathcal{P}_Q$ does not plans better than P_b;
- the risk model can not generalize well on new data/workload, especially in dynamic settings;
- the risk model is under-fitting on the training data owing to loss of features, noisy labels, insufficient training data, bad hyper-parameters or inappropriate training optimizers.

Let Pr_Q be the distribution of all SQL queries occurring for a database. Let $\mathbf{Q}$ be a workload where each query $Q \in \mathbf{Q}$ occurs with the probability Pr_Q. Formally, a learned query optimizer $\mathsf{learned\_opt}$ in our framework generates an execution plan P_r for a query $Q \in \mathbf{Q}$

$$P_r \leftarrow \mathsf{learned\_opt}(\mathcal{P}_Q, M_r)$$

by enumerating candidate plans $\mathcal{P}_Q$ and selecting the best one based on a learned risk model M_r. In practice, the risk model M_r is often trained on a workload $\mathcal{W} \subseteq \mathbf{Q}$, so the regression is often more serious for query $Q \in \mathbf{Q} - \mathcal{W}$.

Our goal is to find other plans to replace P_r with less or no regressions before execution. Here, we assume the plan P_b output by the native traditional query optimizer is in $\mathcal{P}_Q$. A performance elimination method $\mathsf{perf\_elim}$ can be interpreted as a plugin function (in any learned optimizer $\mathsf{learned\_opt}$) which filters out unreliable candidates and selects a different plan P_r'

$$P_r' \leftarrow \mathsf{perf\_elim}(\mathsf{learned\_opt}(\cdot), \mathcal{P}_Q, M_r).$$

Let $\mathcal{R}$ and $\mathcal{B}$ denote the overall performance regression and benefit over all queries in $\mathbf{Q}$, respectively. They are computed as

$$\mathcal{R} = \sum_{Q \in \mathbf{Q}:\, C(P_r) > C(P_b)} (C(P_r) - C(P_b)) \tag{1}$$

and

$$\mathcal{B} = \sum_{Q \in \mathbf{Q}:\, C(P_r) \leq C(P_b)} (C(P_b) - C(P_r)). \tag{2}$$

Let $\mathcal{R}'$ and $\mathcal{B}'$ denote the overall performance regression and benefit by replacing all selected plans P_r with P_r' in Eq. (1) and Eq. (2), respectively. After the replacement, there may exist a positive impact on the performance regression and simultaneously a negative impact on the performance benefit. Obviously, $\mathcal{R} - \mathcal{R}'$ and $\mathcal{B} - \mathcal{B}'$ represent the decline in the performance regression and benefit, respectively. We aim at finding a performance elimination method $\mathsf{perf\_elim}$ that is able to eliminate regressions but brings little impact on the benefits. Formally, our problem is stated as follow:

Performance Regression Elimination Problem

Input: a learned query optimizer $\mathsf{learned\_opt}$ with its risk model M_r trained on a workload $\mathcal{W} \subseteq \mathbf{Q}$ and a parameter $\lambda \geq 0$;

Output: a performance elimination method $\mathsf{perf\_elim}$ such that $\mathcal{R} - \mathcal{R}' + \lambda(\mathcal{B} - \mathcal{B}')$ is minimized.

Here λ balances the decline on the regression $(\mathcal{R} - \mathcal{R}')$ and the loss of the benefit $(\mathcal{B} - \mathcal{B}')$. A large value of λ encourages $\mathsf{perf\_elim}$ to improve the benefit, while a small value of λ emphasizes filtering out risky plans, even at the expense of removing some good plans.

Notably, our problem definition, as well as the proposed solutions, are very general. They are applicable to all learned query optimizers stated above and other ones satisfying our framework. We focus on eliminating the regression of the query optimization problem in this paper. We reserve the extension to other learned tasks on database in future work.

2.3 Analysis of Existing Solutions

In recent times, there has been a significant shift towards enhancing the robustness of learned-based models. For instance, Warper [23] enhances the cardinality estimation model by generating additional queries to update it when data or workload drift is detected. DDUp [19] uses a two-stage sampling procedure to test whether the model should be updated w.r.t. dynamic data. Although these methods are effective, they are post-processing techniques used for model updating. Whereas, our goal is to detect and eliminate regression before query execution.

An intuitive way to eliminate regression is to add more training data. However, this can not guarantee to work well due to two reasons: 1) The plan space size is very large. Adding more training data can not guarantee the model to generalize well on any plan. 2) Some regressions are caused by the limited capacity of models. This under-fitting problem can not be resolved by using more training data. Our experimental results in Section 6.2.3 verify that this method is ineffective.

The literature work [43] tries to apply the ensemble method. It deploys a multi-head LSTM model to learn multiple prediction results for each plan $P \in \mathcal{P}_Q$. All candidate plans P with a large variance in the prediction results are filtered before plan selection. Then, they select the remaining plans with the best average estimated cost to execute. When all candidate plans are filtered, it prefers to execute the plan P_b generated by the traditional optimizer.

The ensemble method has two main drawbacks. First, to improve accuracy, it requires training a large number of models, which is not efficient. Second, its plan filtering strategy can not always reflect the performance of models. A small variance only indicates all models learn similar results for the same plans, but does not ensure the accuracy of the prediction results. A bad plan P may be reserved if all models make the same mistake while a good plan P may be filtered if its cost is only accurately learned by a small number of models. Therefore, the ensemble method can not fundamentally distinguish the models' performance over different plans. By our experiments in Section 6.2, HyperQO has poor performance in eliminating regression. This motivates us to pursue new solutions to resolve the performance regression elimination problem.

3 SYSTEM OVERVIEW

The fundamental reason arising the performance regression is that the risk model can not accurately predict the cost of some plans, due to the lack of generalization ability and/or model under-fitting. Thus, the key to eliminate the performance regression is to identify the learning accuracy for each plan, so that we can filter out all

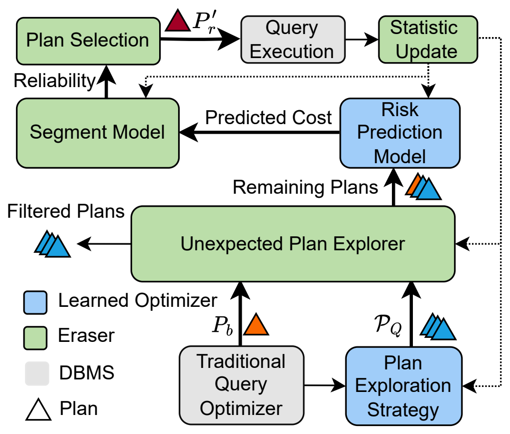


Figure 1: The system architecture and pipeline of `Eraser`.

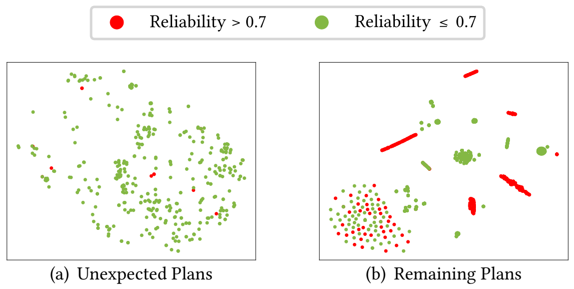


Figure 2: Estimation quality (reliability) of the risk model in Lero on plans of the TPC-H benchmark.

unpromising plans and only reserve other ones with high learning accuracy for plan selection. However, existing learned query optimizers often provide very little knowledge on the learning accuracy of plans. Meanwhile, it is very difficult (or even impossible) to learn the exact estimation accuracy of each plan, which is as challenging as learning the accurate cost of each plan [26, 27, 43].

To this end, we design other tasks which are much easier but could still preserve enough information to identify truly good plans. We design a system, called `Eraser`, to eliminate regression of learn query optimizer. The architecture, as well as its pipeline, is shown in Figure 1.

Notably, `Eraser` could be deployed on top of any learned optimizer, as long as it satisfies our proposed framework in Section 2, to eliminate its performance regression while reserving the performance improvement. Given a query Q, we first collect the plan P_b generated by the traditional query optimizer and candidate plans $\mathcal{P}_Q$ produced by the plan exploration strategy in the learned query optimizer. Later, `Eraser` adopts a two-stage strategy to identify another plan P'_r to execute. Its main idea is discussed as follows.

Main Idea of the Strategy in `Eraser`. We have some basic observations on the estimation quality of risk models in learned query optimizers. Specifically, in Figure 2, we employ t-SNE [39] to map the estimation quality of the risk model in Lero [46] on plans of the TPC-H benchmark [6] onto a two-dimensional space. The green and red points indicate plans with low and high estimation accuracy (defined in Section 5.3), respectively. The phenomenon on other learned query optimizers and benchmarks are similar. We find that:

1) Figure 2(a) shows the performance on plans for queries $Q \in \mathcal{Q} - \mathcal{W}$. We call these plans *unexpected plans* as they contain feature values not occurring in the training data. Obviously, the risk model performs badly on most of the unexpected plans, due to its low generalization ability. Therefore, in the first stage, we design an ***unexpected plan explorer*** to systematically investigate the space of all unexpected plans to quantitatively evaluate the performance of the risk model. Based on it, we could filter all highly risky plans in $\mathcal{P}_r$ at the front in a coarse-grained manner, which largely reduces the burden of downstream procedures.

2) Figure 2(b) illustrates the performance on remaining plans for queries Q falls into $\mathcal{W}$ and a small portion of unexpected plans where the risk model could generalize well. At this time, the performance of the risk model is highly *skewed*. The plans naturally form different clusters. Some of them contain purely accurate or inaccurate plans, but some clusters, e.g., the cluster in the left bottom, contain a mixture of both accurate and inaccurate plans. This is because the risk model does not have enough capacity to fit well on all plans. It is under-fitting for some subspace of plans. To distinguish them, we apply a ***segment model*** to cluster plans in a more fine-grained manner. Meanwhile, we associate each cluster of plans with an interval of *reliability*, which reflects the range of the estimation quality of the risk model on these plans.

Based on the reliability interval, we could further filter some unpromising plans with low learning accuracy. We design a plan selection method in `Eraser` to select the final execution plan P'_r. This method could balance the benefit of improvements and the risk of regressions. For example, in a conservation scenario, we could only reserve plans with high reliability, which may miss some improvement opportunities but could reduce the performance regression to a very low level. On the contrary, in an aggressive scenario, we could relax the risk constraint to pursue more possible improvements. After the plan P'_r is executed by the query execution engine, we collect its execution statistics to periodically update the plan exploration strategy and risk model in the original learned query optimizer, and simultaneously the unexpected plan explorer and segment model in `Eraser`.

In the following content, we introduce the technical details of the unexpected plan and the segment model (together with the plan selection method) in Section 4 and Section 5, respectively.

4 UNEXPECTED PLAN EXPLORER

In this section, we present the details on investigating the space of unexpected plans. We propose a method that hierarchically divides these unexpected plans into a number of subspace according to the domain of features. Then, we select a small number of unexpected plan subspace and generate some plans in each subspace. Based on these plans, we could know how well the risk model performs in any unexpected plan subspace. In the following, Section 4.1 introduces the basic plan encoding method. Section 4.2 describes the main framework of our space division method. Then, Section 4.3 and Section 4.4 discuss two key techniques applied in our framework.

4.1 Plan Encoding Method

In this paper, we discuss our method using the SQL query Q in the following form:

$$\text{SELECT} \; * \; \text{FROM} \; T_1, T_2, \ldots, T_m$$
$$\text{WHERE} \; J_1, J_2, \ldots J_{m-1} \; \text{AND} \; E_1, E_2, \ldots, E_n.$$

Here each $T_{1 \le i \le m}$ refers to a table in the database, each $J_{1 \le j \le m-1}$ stands for a join relation, e.g., equal or non-equal join, between any two columns in tables, and each $E_{1 \le \ell \le n}$ represents a filtering predicate on a column. Notice that, we do not consider any feature related to the projection columns and nested SQL queries. However, our proposed encoding and division methods could be easily extended to support these queries.

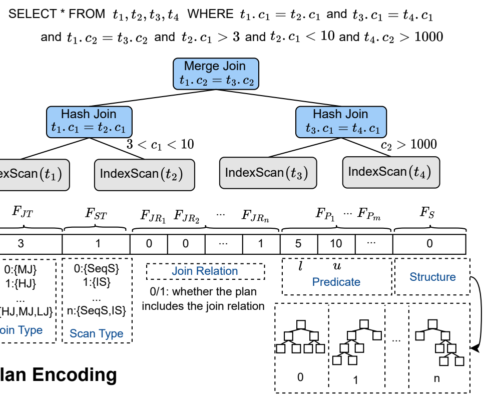
A physical plan P of Q can be represented as a binary tree structure, as the example shown in Figure 3. In the plan tree, each leaf node has a scan operation on a column C in a table with its filtering predicate and each inner node has a join operation with the join relation across two tables. Notably, the information of query Q is losslessly contained in the plan P.

In the literature work on learned query optimizer [26, 28, 42, 43, 46], the query plans are featured as vectors and fed into the risk model for cost prediction. Our goal is to evaluate the influence of different feature values on the accuracy of the risk model. To this end, we try to encode each query plan with the widely used features in learned query optimizers. Specifically, we consider the following features:

- *join and scan type*: two categorical features indicating the types of all join and scan operators used in the query plan. For example, assume that the DBMS supports three types of join operators, namely merge_join (MJ), hash_join (HJ) and nested_loop_join (LJ), then the value of the join type has seven different values, where each corresponds to a non-empty subset of {MJ, HJ, LJ }. For the plan shown in Figure 3, the value of join type is {MJ, HJ}.
- *join relations*: a vector encoding the existence of join relations occurred in the query plan. The set of all possible join relations across any two tables are provided by users or found by an auto-exploration method proposed in [45]. We use a binary variable to encode the existence of each join relation in the vector.
- *filtering predicate*: a vector encoding the filtering condition on each column (attribute). Specifically, we represent the predicate on each column C in a canonical form $l \le C \le u$ and record the two endpoints l and u in the vector. Let lb and ub denote the lower and upper bound of C, respectively. Other forms of the predicates could be equivalently converted into this form as

$$(C \le u) \to (lb \le C \le u), (C \ge l) \to (l \le C \le ub),$$
$$(C < u) \to (lb \le C \le u - \epsilon), (C > l) \to (l + \epsilon \le C \le ub),$$
$$(l < C < u) \to (l + \epsilon \le C \le u - \epsilon), \text{ where } \epsilon \to 0^+.$$

  In our implementation, we discretize the domain of each continuous attribute to a number of bins, so ϵ could be set to a value smaller than the bin width.
- *structure*: a categorical variable indicating the shape of the plan tree. Each value corresponds to a specific form of the plan ignoring the operator type and filtering predicates on all nodes (as they have been encoded in other features). For example, Figure 3 lists several possible structures, e.g., bushy tree, left-deep and right-deep, on plans joining 4 tables.



SELECT * FROM t_1, t_2, t_3, t_4 WHERE $t_1.c_1 = t_2.c_1$ and $t_3.c_1 = t_4.c_1$ and $t_1.c_2 = t_3.c_2$ and $t_2.c_1 > 3$ and $t_2.c_1 < 10$ and $t_4.c_2 > 1000$
Merge Join $t_1.c_2 = t_3.c_2$
Hash Join $t_1.c_1 = t_2.c_1$
Hash Join $t_3.c_1 = t_4.c_1$
$3 < c_1 < 10$
$c_2 > 1000$
IndexScan(t_1)
IndexScan(t_2)
IndexScan(t_3)
IndexScan(t_4)
F_{JT} F_{ST} F_{JR_1} F_{JR_2} ··· F_{JR_n} F_{P_1} ··· F_{P_m} F_S
3 1 0 0 ··· 1 5 10 ··· 0
0:{MJ} 1:{HJ} ... n:{HJ,MJ,LJ} Join Type
0:{SeqS} 1:{IS} ... n:{SeqS,IS} Scan Type
Join Relation
0/1: whether the plan includes the join relation
l u Predicate
Structure
0 1 ... n
Plan Encoding

Figure 3: An example of plan encoding.

A complete example of our plan encoding is shown in Figure 3. Notice that, our plan encoding method is independent of the risk model. Meanwhile, it is flexible enough to be extended to support other new features specified by users.

4.2 Division Method Framework

Next, we consider how to investigate the performance of the risk model over unexpected plans. To balance efficiency and accuracy, we propose a very general framework. It hierarchically divides the plan space into a number of subspace and generates plans in some representative subspace to evaluate the model performance.

Without loss of generality, let $F_1, F_2, \ldots, F_n$ denote a number of features of query plans encoded by us. For each feature F_i, we denote its domain as D_i. Let S_i and U_i denote all values of F_i occurring and unseen in the training workload $\mathcal{W}$, respectively. For a plan P, if P contains any unseen value $d_i \in U_i$ of any feature F_i, we call P an unexpected plan. Obviously, the feature space $\mathcal{U}$ of all unexpected plans is $\mathcal{U} = (D_1 \times D_2 \times \cdots \times D_n) - (S_1 \times S_2 \times \cdots \times S_n)$.

Each point $p = (d_1, d_2, \ldots, d_n) \in \mathcal{U}$ refers to all plans having value d_i for feature F_i. Note that, these plans share the same encoding vector so they are difficult to distinguish for the learned risk model. To investigate the model performance on any unexpected plan, we need to evaluate the performance of the risk model on each point. However, this is very costly and unrealistic as the unexpected plan space contains $|\mathcal{U}| = \prod_i |D_i| - \prod_i |S_i|$ points[1]. This size grows exponentially w.r.t. the number of features and $|D_i - S_i|$ is very large for some attributes, i.e., filtering predicates.

To address this problem, we design a new algorithm by selecting a small number of representative unexpected plans. Our method arises from two fundamental observations:

- First, we observe that if the risk model in the learned query optimizer can not perform well on plans with only one single unseen value $d_i \in U_i$ of feature F_i, it is highly likely to perform worse on plans with unseen values on more features. This is reasonable as the bad performance implies that the model does not acquire enough knowledge to process value d_i on F_i, even with the help of other features F_j where $j \neq i$. At this time, providing other unseen values d_j to the model would certainly not contribute, but degrade its performance.

[1]For continuous features such as the filtering predicate, $|D_i|$ refers to the bin size.

Based on this, we could evaluate all points $p \in \cup_i(S_1 \times \cdots \times S_{i-1} \times U_i \times S_{i+1} \times \cdots \times S_n)$ in the unexpected plan space having only one unseen feature value. Their evaluation results could help us to filter other unexpected plans with bad performance. The number of such points is only $\sum_i |U_i|$, which is much less than the number of all points $(\prod_i |D_i| - \prod_i |S_i|)$ in the space.

- Second, as illustrated in Figure 2(a), the risk model is likely to have similar performance for nearby value $d_i, d'_i \in D_i$ of feature F_i, especially for the continuous feature. This is because similar plans may have similar execution time, so as the outputs of the prediction model. This implies that we could group nearby unseen values together to further reduce the evaluation cost.

Algorithm Description. Based on these observations, we present our method in the Algorithm Plan_Space_Division. For each feature F_i with its unseen domain U_i, we split U_i into a number of smaller and disjoint subset $U_i^1, U_i^2, \ldots, U_i^k$ and call the Procedure Recur_Split to recursively divide each subset. We apply three methods for division. For domain U_i with categorical value, if $|U_i|$ is smaller than a threshold, we set each subset U_i^j to contain a singleton value in U_i; otherwise, we randomly split each value U_i to two subsets U_i^1 and U_i^2. For domain U_i with (discretized) continuous value, we binary split U_i into two subsets U_i^1 and U_i^2 with equal size. Obviously, the division of U_i forms a hierarchical structure. The users could control the splitting granularity to balance the evaluation efficiency and accuracy.

The splitting process terminates if $|U_i^j|$ is smaller enough than a user-specified threshold. For each U_i^j without further splitting, we then generate a number of plans falling into this subspace, namely $S_1 \times \cdots \times S_{i-1} \times U_i^j \times S_{i+1} \times \cdots \times S_n$. Then, we evaluate the risk model performance using these plans. If the model can not perform well, we mark all points p in the unexpected plan space $\mathcal{U}$ whose feature value of F_i falling into U_i^j, namely $p \in D_1 \times \cdots \times D_{i-1} \times U_i^j \times D_{i+1} \times \cdots \times D_n$, as imprecise (by our first observation). Otherwise, we mark all points p in the evaluated subspace as precise.

Note that, as we only need to examine the performance of risk models on each subspace, we do not need to generate a large number of plans. In our experiments, we show that we only need to generate a small number (e.g., several hundred) of queries to evaluate model performance. In the following, we introduce our implementations on generating plans for evaluation in each subspace (in Section 4.3) and evaluating the model performance (in Section 4.4).

Remarks on Updating. Notably, our space division method is independent of the risk model. As a result, when the risk model updates, including retraining or fine-tuning, we do not need to regenerate plans in each subspace. Instead, we could keep all generated plans and re-evaluate the model performance on each subspace to mark it accordingly.

4.3 Plan Generation Method

We design a method to manually generate a number of new plans having unseen value on one feature in four steps. Remarkably, we use the *hints* on DBMS to control the join/scan operators used in the plan and the join order between tables. They are supported by

```
Algorithm Plan_Space_Division($F_1, \ldots, F_n, D_1, \ldots, D_n, U_1, \ldots, U_n$)
1: for each $1 \le i \le n$ do
2:     $S_i \leftarrow D_i - U_i$
3: end for
4: for each feature $F_i$ do
5:     Recur_Split($F_i, U_i, D_1, \ldots, D_n, S_1, \ldots, S_n$)
6: end for

Procedure Recur_Split($F_i, U_i, D_1, \ldots, D_n, S_1, \ldots, S_n$)
1: if $|U_i|$ is smaller enough according to feature $F_i$ then
2:     generate plans falling into $S_1 \times \cdots \times S_{i-1} \times U_i^j \times S_{i+1} \times \cdots \times S_n$
3:     evaluate model performance using generated plans
4:     if the model perform bad then
5:         mark all points $p \in D_1 \times \cdots \times D_{i-1} \times U_i^j \times D_{i+1} \times \cdots \times D_n$ as imprecise
6:     else
7:         mark all points $p \in S_1 \times \cdots \times S_{i-1} \times U_i^j \times S_{i+1} \times \cdots \times S_n$ as precise
8:     end if
9: else
10:    divide $U_i$ into disjoint subset $U_i^1, U_i^2, \ldots, U_i^k$
11:    for each $U_i^j$ where $1 \le j \le k$ do
12:        Recur_Split($F_i, U_i^j, D_1, \ldots, D_n, S_1, \ldots, S_n$)
13:    end for
14: end if
```

popular DBMS such as PostgreSQL, MySQL and SQL Server. We could set the hint command to enable/disable certain operators, e.g., enable only merge_join and hash_join, or specify the join order, e.g., join T_a with T_b then T_a with T_c, before the plan generation. The process is as follows:

- First, we obtain a join form as the template for query generation. For each join relation F_i, if the generated unexpected plan space requires the only unseen value occurring in d_i of F_i, we pick its value d_i from U_i. Otherwise, the value d_i of F_i must contain seen values, so we randomly sample $d_i \in S_i$. We continue the following steps if the obtained join form is valid across tables.
- Second, we attach filtering predicates on each column (attribute). For each filtering predicate F_i, we also pick d_i from U_i if F_i is the required unseen value. Otherwise, we randomly sample $d_i \in S_i$ such that the two endpoints l and u satisfy $l \le u$. Then, we obtain a valid query Q for further plan generation.
- Third, we specify the structure of the generated plan P of Q. In similar, we pick a possible structure shape d_i from U_i if the structure F_i is the required unseen value. Otherwise, we randomly sample a seen structure shape $d_i \in S_i$ for Q. Then, we randomly select a join order between tables in Q according to d_i and set the hint according to the join orders into DBMS.
- Forth, we restrict the set of available join/scan types. If the join/scan type F_i is required to be unseen, we pick d_i from U_i. Otherwise, we randomly select a seen set $d_i \in S_i$ of join/scan types for P. Then we set the hint on available operators into DBMS based on d_i.

After that, we ask the native query optimizer to generate the plan P for query Q and collect such plans for model evaluation.

4.4 Model Performance Evaluation

After obtaining a number of generated unexpected plans in each subspace, we then consider how to evaluate the performance of the risk model. In the literature work, the risk model could be either a pointwise regression model [26, 43] or pairwise a classification model [13, 46]. We process it in different ways.

If the risk model is a pointwise model, it takes a plan P as input and predicts its estimated cost $\widehat{C}(P)$ to approach the exact cost $C(P)$.

At this time, we define the relative error ratio $e(P)$ of the plan P as

$$e(P) = \min(\frac{\widehat{C}(P)}{C(P)}, \mathrm{UB}). \quad (3)$$

Here UB is an upper bound to prevent severe prediction bias of a part of plans to dominate the average error ratio. We set it to 2 in our experiments. We compute the average ratio $\overline{e}(P)$ of all unexpected plans generated for the subspace. We always have $\overline{e}(P) \in [0, \mathrm{UB}]$. We mark the subspace as precise if $\overline{e}(P)$ is less than a threshold α and imprecise otherwise. Notice that, the hyper-parameter α is tuned to be proportional to the input parameter λ, as large α could filter out more risky plans.

When the risk model is a pairwise model, it takes a pair of plans P, P' and outputs a binary label to indicate which plan is better. For such models, we directly collect all pairs of generated plans and compute $e(P)$ as the proportion of pairs where the risk model could accurately find the better plan. We then mark the subspace using a threshold α in the same way as the regression model.

5 SEGMENT MODEL AND PLAN SELECTION

After filtering all unexpected plans where the risk model is highly likely to perform badly, we next discuss how to process the remaining plans in the whole place space in this section. At this time, we need to process each plan in a more fine-grained manner to filter unpromising plans. By our observations, the performance of the risk model is different for different regions in the plan space. Therefore, we design a segment model to cluster plans according to model performance and associate each cluster with the reliability interval for plan selection. We introduce the segment model design and training details in Section 5.1 and Section 5.2, respectively. Then, Section 5.3 presents the method for plan selection.

5.1 Segment Model Design

We discuss how to design the segment model in this subsection, including the loss function and the model structure.

Loss Function. Formally, let s be a possible segment model that divides the plan space into multiple non-overlapping subspace $\mathcal{G} = \{G_1, G_2, \ldots, G_k\}$. Recall that, $\mathcal{R}$ and $\mathcal{B}$ (or $\mathcal{R}'$ and $\mathcal{B}'$) denote the regression and benefit before (or after) applying our segment model and plan selection method (discussed in Section 5.3). By Section 2.2, our goal is, after clustering remaining plans, we could balance the risk of regressions and the benefit of improvements in the plan selection stage. To this end, we train the segment model with the following loss function

$$L(s) = \log(\mathcal{R}' - \mathcal{R}) + \lambda(\mathcal{B} - \mathcal{B}')) + \lambda_1 |\mathcal{G}| + \lambda_2 \sum_i \max(\sigma - |G_i|, 0). \quad (4)$$

Here the first term is our goal on minimizing the performance regression elimination problem (see Section 2.2). The second term penalizes the clustering granularity on the number of groups. The third term penalizes the number of plans in each group where $\sigma > 0$ is a hyper-parameter to restrict the minimum number of plans in each group. We do not encourage too many clusters and too few plans in each group to avoid over-fitting.

Model Structure. We have several fundamental requirements for the segment model. First, it should be lightweight as the segment model would be frequently retrained when the risk model is updated. Second, the behavior of the segment model should be deterministic, and better to be explainable. This is because the behavior of risk model, which rely on deep neural networks, is often uncertain [40]. As we try to eliminate its performance regression, it is better not to bring additional uncertainty w.r.t. downstream models. Otherwise, it is difficult to analyze whether the additional error of the segment model would reduce or enlarge the regression. To this end, we pursue a traditional statistical model rather than a deep model, whose training cost is often higher and behavior is often highly uncertain.

We note that, our learning task resembles the classification process of the decision tree, which recursively splits the training data according to their labels using different features. We borrow this idea to design our segment model with two critical differences.

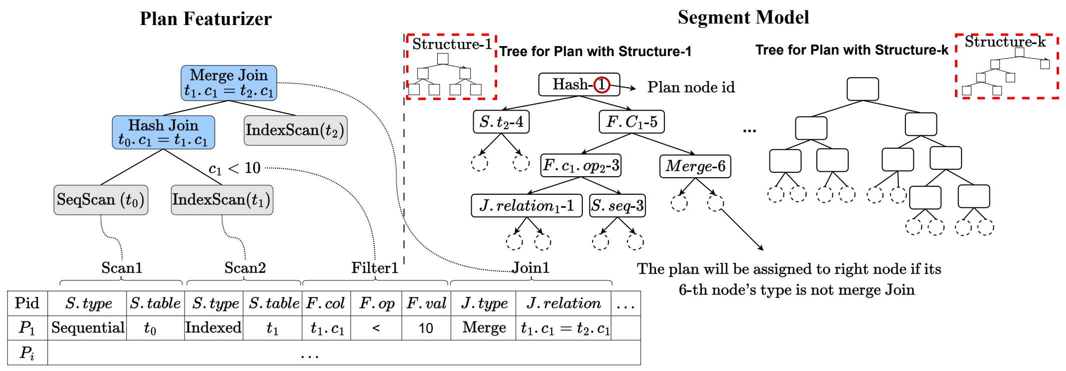
First, training our segment model is an unsupervised task. The decision tree is split to minimize the classification error. Whereas, our splitting criteria is to minimize the loss function in Eq. (4). The details are elaborated in the following Section 5.2.

Second, the structure of the plan tree is very diversified, which is difficult to be encoded as a vector having a fixed length. To address this problem, existing deep models [26, 28, 46] encode the feature of each node and then generate the tree-level encoding using convolution operation [29]. Obviously, this method is not applicable to our segment model. Another possible way is to compress all node-level vectors into a tree-level vector using bitwise sum or mean value [7]. However, this would lose the structural information. In our design, we maintain a forest where each tree focuses on clustering plans having a specific form of structure ignoring the operator type and filtering predicates on all nodes. In such way, the learning hardness of the segment model is further reduced, as each tree only needs to process plans based on their features w.r.t. operations, tables and predicates.

Figure 4 (the right part) illustrates an example of our segment model, where we list several possible plan structures. The feature vector of each plan is obtained by concatenating the encoding vector on all nodes. Specifically, for all inner node with join operation, we encode the join operator type and the join relation as categorical variables. For all leaf nodes with scan operation, we encode the scan operator type and the scanned table as categorical variables. Meanwhile, we also encode the filtering predicate, which includes the column, the operator type and the filtering value. We show an example of the plan encoding in the left part of Figure 4.

5.2 Model Training

We train the segment model using the set of plans $\mathcal{P}_S$. It contains all candidate plans for the queries Q in the training workload $\mathcal{W}$, i.e., $\bigcup_{Q \in \mathcal{W}} (\mathcal{P}_Q \cup \{P_b\})$. Thus, $\mathcal{P}_S$ is a snapshot of the remaining plans filtered by the unexpected plan explorer. For each kind of plan structure, the tree based segment model is constructed in the same manner. We present the model training method in the Algorithm Model_Train. At the very beginning, we collect all plans $\mathcal{P}_N \subseteq \mathcal{P}_S$ with this kind of structure and maintain it in the root node. Then, the algorithm works in a recursive manner. Each time it splits all plans in the parent node into two children using the best split feature value. We stop the splitting process unless the stopping condition is satisfied.



Pid	$S.type$	$S.table$	$S.type$	$S.table$	$F.col$	$F.op$	$F.val$	$J.type$	$J.relation$	...
P_1	Sequential	t_0	Indexed	t_1	$t_1.c_1$	<	10	Merge	$t_1.c_1 = t_2.c_1$	
P_i	...									

Figure 4: The example of the segment model. The left part is the plan featurizer. The right part is a forest, where each tree represents the clustering model for a specific structure.

Specifically, let f_i denote the i-th feature in the plan's encoding vector. Let $f_{i,j}$ denote the j-th value of the feature f_i. For any node N with the set of plans $\mathcal{P}_N$, we split $\mathcal{P}_N$ into $\mathcal{P}_L$ and $\mathcal{P}_R$ on the left and right child according to $f_{i,j}$, respectively. If f_i is a categorical variable, $\mathcal{P}_L$ contains all plans having value $f_{i,j}$ on feature f_i. Otherwise when f_i is a continuous variable, $\mathcal{P}_L$ contains all plans whose feature value of f_i is no more than $f_{i,j}$. Then we set $\mathcal{P}_R = \mathcal{P}_N - \mathcal{P}_L$.

To minimize the loss function, we select the best splitting feature value $f^*_{i,j}$ based on the greedy strategy. For each splitting feature value $f_{i,j}$, we could obtain a possible tree structure after applying $f_{i,j}$ for splitting. Based on this tree structure, we could select another execution plan P'_r to replace the original execution plan P_r using the plan selection method (discussed in Section 5.3). Then, we could compute the new regression $\mathcal{R}'$ and the new benefit $\mathcal{B}'$ using all plans P'_r. We select the best splitting feature value $f^*_{i,j}$ that minimizes the loss function defined in Eq. (4). To avoid over-fitting, we terminate the splitting process when the number of plans on the current node is less than a pre-defined threshold, typically, 5% of the training data size.

Remarkably, since the three features, i.e., the column, operator and filtering predicate, for the filtering operator are dependent. That is, different columns (attributes) would have different filtering operators and predicates. We add a constraint that the filtering operators and predicate can only be selected as the splitting condition if the corresponding column has been applied for the split in the ancestors.

Remarks on Updating. We retrain our segment model if the risk model is rebuilt. Otherwise, we could update it using executed plans in an incremental manner. Specifically, for new plans P, we insert P into the corresponding leaf node $\mathcal{P}_L$ according to the splitting condition. If the size of $\mathcal{P}_L$ exceeds the limit, we further split $\mathcal{P}_L$ into smaller groups until the stopping condition is satisfied.

5.3 Plan Selection Method

Based on the segment tree model, we introduce how to select plans in this subsection. According to how the risk model is designed in existing learned query optimizer [13, 26, 28, 42, 43, 46], we process it in different ways.

When the risk model is a pointwise regression model, for each candidate plan P, we try to find the corresponding tree model for P according to its plan structure. If we have not trained model for

```
Algorithm Model_Train($f_1, f_2, \ldots, f_t, \mathcal{P}_N$)
 1: for each $f_{i,j}$ do
 2:     split $\mathcal{P}_N$ into $\mathcal{P}_L$ and $\mathcal{P}_R$ by $f_{i,j}$
 3:     compute the loss using the current tree structure
 4: end for
 5: find the feature value $f^*_{i,j}$ with the minimum loss
 6: get $\mathcal{P}_L$ and $\mathcal{P}_R$ by $f^*_{i,j}$
 7: set $\mathcal{P}_L$ and $\mathcal{P}_R$ to be the left and right child of $\mathcal{P}_N$
 8: if $|\mathcal{P}_L|$ is less than the pre-defined threshold then
 9:     set $\mathcal{P}_L$ to be a leaf node
10: else
11:     Model_Train($f_1, f_2, \ldots, f_t, \mathcal{P}_L$)
12: end if
13: if $|\mathcal{P}_R|$ is less than the pre-defined threshold then
14:     set $\mathcal{P}_R$ to be a leaf node
15: else
16:     Model_Train($f_1, f_2, \ldots, f_t, \mathcal{P}_R$)
17: end if
```

such structure, it implies we have very limited knowledge for it, so we safely skip plan P to avoid risk or accept plan P to attain potential benefit. Otherwise, in this model, there must exist a leaf node having plans $\mathcal{P}_L$ such that P is assigned into $\mathcal{P}_L$. We define a reliability value $r(P)$ for each plan P. Obviously, the relative error ratio defined in Eq. (3) could reflect the quality of the estimation results. Therefore, we directly define $r(P) = \widehat{C}(P)/C(P)$. Then, we filter plans according to the reliability interval of plans in $\mathcal{P}_L$.

Specifically, let $d(\mathcal{P}_L) = \max_{P \in \mathcal{P}_L} r(P) - \min_{P \in \mathcal{P}_L} r(P)$ denote the width of the reliability interval of plans in $\mathcal{P}_L$. If $d(\mathcal{P}_L)$ is smaller than a user-specified threshold β, it indicates the reliability of plans in $\mathcal{P}_L$ are very similar. By our observations in Figure 2(b), at this time, plans in $\mathcal{P}_L$ have similar levels of accuracy, i.e., they may all be accurately or inaccurately learned together. Therefore, we trust this reliability value. Let $\bar{r}(\mathcal{P}_L)$ be the average reliability value of plans in $\mathcal{P}_L$. We correct the predicted cost $C(P)$ to $C'(P) = C(P)/\bar{r}(P)$. Otherwise when $d(\mathcal{P}_L)$ is larger than β, it indicates the range of the reliability value is not tight. At this time, it implies that this group of plans may not have similar accuracy levels, such as the cluster in the left bottom in Figure 2(b). Therefore, we filter this candidate plan P as we do not have confidential information to correct its estimated cost. After the correction, we select all remaining plans P with the minimum $C'(P)$ to execute. Users could adjust the threshold β to balance the potential benefit and regression risk.

When the risk model is a pairwise classification model, it takes a pair of candidate plans P, P' and outputs which plan is better in terms of their cost. At this time, if we can not find the trained model for any of them, we could also skip comparing P and P' to avoid risk. Otherwise, we find the segment model, as well as the leaf node $\mathcal{P}_L$ and $\mathcal{P}'_L$ for plans P and P', respectively. Then, we collect all pair of plans (P_1, P_2) such that $P_1 \in \mathcal{P}_L$ and $P_2 \in \mathcal{P}'_L$. Let $r(\mathcal{P}_L, \mathcal{P}'_L)$ denote the portion of pairs that the risk model could accurately find the better plan. $r(\mathcal{P}_L, \mathcal{P}'_L)$ could also indicate the confidence of the risk model on these plans. In similar, we trust the risk model if $l(P, P')$ is larger than a user-specified threshold β. At this time, we think plan P surpasses P' if the risk model predicts plan P to be better and vice versa. Otherwise, we do not trust the risk model and ignore the comparison results between P and P'. Finally, the plan P that surpasses the most number of other plans is selected to execute.

6 EVALUATION RESULTS

In this section, we conduct experiments to comprehensively investigate the performance of our `Eraser` system. We make our implementation of `Eraser` open-source[2]. Specifically, our experiments aim at answering the most crucial questions as follows:

- When `Eraser` is deployed on top of the existing learned query optimizer, how much performance regression could it eliminate, and how much impact it may cause to the benefit? (in Section 6.2)
- Could `Eraser` adapt well in dynamic settings? (in Section 6.3)
- How much contribution does each component in `Eraser` make in eliminating performance regression? (in Section 6.4)
- What is the impact of the input parameter λ? (in Section 6.5)
- Could `Eraser` be applied for distributed databases? (in Section 6.6)

6.1 Experimental Setup

Baselines. We use three representative learned query optimizers in our experiments. Specifically, **HyperQO** [43] and **Lero** [46] are two learned query optimizers using pointwise and pairwise risk models, respectively. HyperQO applies the ensemble method to eliminate regression. They have been shown to perform better than Bao [26]. Besides, Lero is shown to perform better than Balsa [42]. Therefore, we do not apply Bao, Balsa and Neo [28] (which performs even worse than Bao) in our experiments. **PerfGuard** [13] is a learned query optimizer which supports any plan generation strategy. We also use Lero's plan exploration strategy to generate plans for PerfGuard. We implement HyperQO and Lero using their source code in [1] and [2], respectively and implement PerfGuard by ourselves. All of them are deployed on the native query optimizer of **PostgreSQL**, which serves as the basic traditional query optimizer. For each learned query optimizer, we deploy `Eraser` on top of it. We denote the resulting query optimizer as **HyperQO-`Eraser`**, **Lero-`Eraser`** and **PerfGuard-`Eraser`**, respectively.

Benchmarks. We evaluate the performance of all query optimizers on four benchmarks widely used in the literature [26, 28, 43, 46]. We summarize their statistical information in Table 1. Each benchmark contains a number of tables and query templates with various types of joins. We generate a training and testing workload for each benchmark. In each workload, each time we randomly pick a query template, and then attach some randomly generated predicates to it. Notably, for TPC-H and TPC-DS, we select 14 and 49 templates in

[2]https://github.com/duoyw/Eraser

Table 1: Statistics of benchmarks used in experiments.

Statistic	IMDB	STATS	TPC-H	TPC-DS
Source	[22]	[9]	[6]	[5]
# of tables	21	8	8	24
# of query templates	113	146	14	49
# of training queries	1,000	1,000	700	4,900
# of testing queries	113	146	140	490

all the templates for query generation. The other templates contain complex features such as nested SQL queries or views that are not supported by HyperQO or Lero. The TPC-DS benchmark is mainly used for experiments on distributed database (in Section 6.6). The remaining ones are used for experiments on PostgreSQL (in Section 6.2 to Section 6.5).

Evaluation Methods. Following [26, 46], we evaluate all learned query optimizers in two scenarios. In the first scenario, we evaluate their stable performance. At this time, the learned query optimizers are trained on a number of training queries. Then we investigate their performance on the test workload with the stable learned models. In the second scenario, we simulate a real-world environment to test its online performance. At this time, all learned query optimizer starts with randomly initialized models. Then, the learn query optimizer processes each training query online one by one and retrains its model using all observed queries after seeing every 100 queries.

Parameters. For HyperQO, Lero and PerfGuard, we use the same default hyper-parameters in the original paper. For our `Eraser`, we set the input parameter $\lambda = 0.8$ on all benchmarks. This choice makes the best trade-off between regression and improvement to attain the lowest overall execution time. In each benchmark, we generate 200 queries to evaluate model performance to filter unexpected plans. We tune the hyper-parameters α for filtering unexpected plans and β for filtering unreliable plans in `Eraser` using grid search to attain the best overall performance.

Environments. All experiments are conducted on a Linux machine with an Intel(R) Xeon(R) Platinum 8163 CPU running at 2.5 GHz, 96 cores, 768GB DDR4 RAM and 2TB SSD. Eight NVIDIA Tesla V100-SXM2 GPUs are equipped for model training and inference. The version of PostgreSQL is 12.1, which is configured with 4GB shared buffers. For the experiments on the distributed database, we also install Spark 3.3 [44].

6.2 Performance of `Eraser`

We evaluate the performance of all learned query optimizers, especially the ones with `Eraser`, by the two evaluation methods in Section 6.1.

6.2.1 *Performance with Stable Models.*

We train each learned query optimizer on the 25%, 50%, 75% and 100% data of each training workload and then test it on the test workload, respectively. This could reflect the generalization ability of learned query optimizers on queries with unseen feature values. Figure 5 shows the average execution time of all queries in the test workloads in three benchmarks. We have the following observations:

1) Performance regressions commonly occur in learned query optimizers, especially when the test workloads contain queries with unseen feature values. For example, Lero, HyperQO and PerfGuard

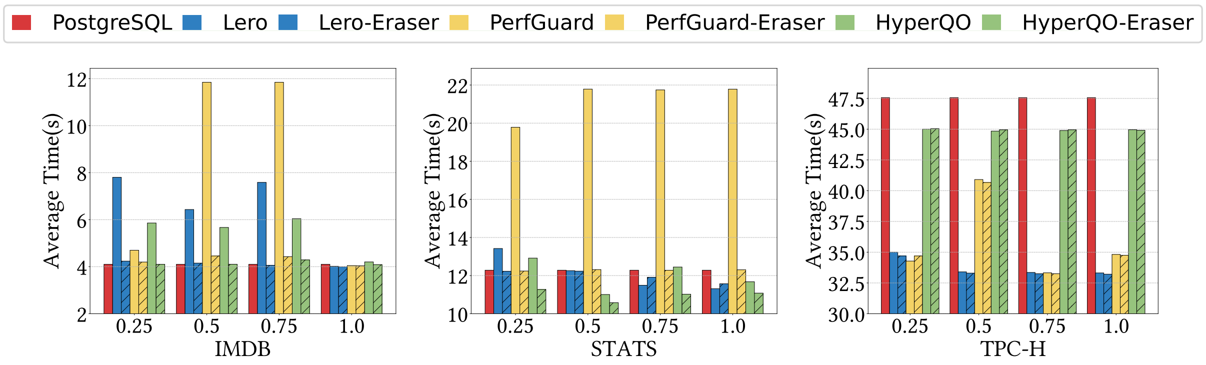
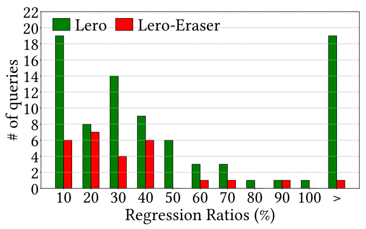
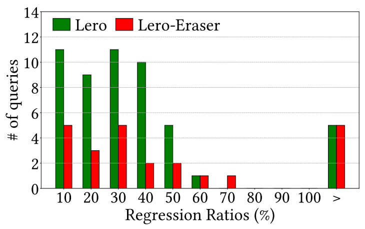


Figure 5: Performance of learned query optimizer with stable models on three benchmarks.

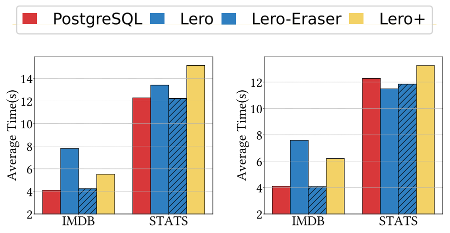


(a) IMDB benchmark with 25% training data



(b) IMDB benchmark with 100% training data

Figure 6: Performance of Lero-Eraser on queries with different levels of regression.



(a) Training on 25 % Data (b) Training on 75 % Data

Figure 7: Performance when training model with more data.

all perform much worse than PostgreSQL unless they witness 100% of the training data on IMDB. This indicates that eliminating the performance regression is crucially important to improve the stability of learned query optimizer. Meanwhile, the ensemble method in HyperQO can not ensure to eliminate regression. The reasons are analyzed in Section 2.3.

2) By deploying our Eraser, almost all of the performance regressions are eliminated. In all cases when the original learned query optimizer performs worse than PostgreSQL, Eraser could help to improve its performance to be comparable, or slightly better than PostgreSQL. This indicates that Eraser could filter most of the unpromising plans which are falsely predicted and selected by the risk models in learned query optimizers.

3) When the learned query optimizers perform well, Eraser brings very little negative impact on the performance. In all cases when the original learned query optimizer performs better than PostgreSQL, its execution time makes little difference, sometimes even better, when deployed with Eraser. This indicates that Eraser could learn enough information to match our desired goal.

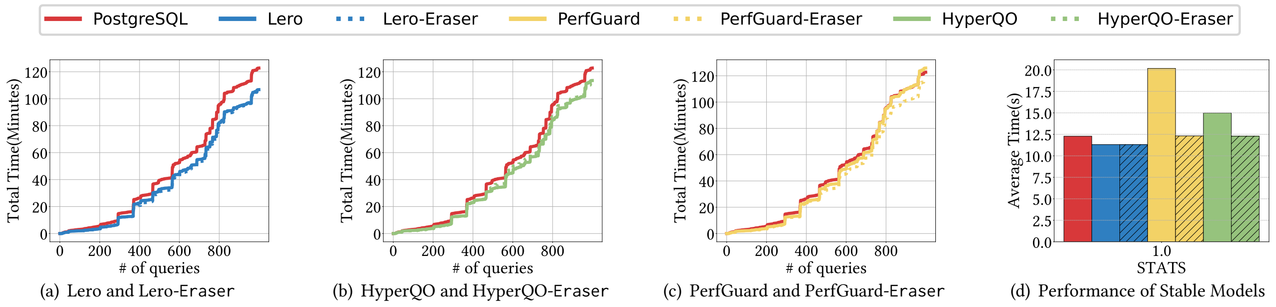
6.2.2 Analysis of Regression Elimination. To present more details, we present the effects of Eraser on queries with different levels of regression. On the IMDB benchmark, we divide all test queries Q with regressions according to the ratio of the regression time, i.e., $(C(P_r) - C(P_b))/C(P_b)$. The results of Lero and Lero-Eraser trained on 25% and 100% data in IMDB are shown in Figure 6. We have two key insights:

1) After applying Eraser, the number of regression queries has a significant decrease, i.e., from 84 to 27 in Figure 6(a) and from 52 to 24 in Figure 6(b). In some ideal cases, for example in Figure 6(a), Eraser eliminates 100% and 94% of regression queries whose regression ratio is between $(50\%, 60\%]$ and $(100\%, \infty)$, respectively.

2) The behavior of Eraser is different on models trained using different volumes of data. In Figure 6(a), the model in Lero is trained on the part of the training data, so it witnesses a number of queries having unseen feature values in the test workload and causes heavy regression. At this time, Eraser filters these unexpected plans with high risks (such as the right part of Figure 6(a)). However, when the model is trained on all data and the overall performance of Lero is better than PostgreSQL (in Figure 6(b)), the number of filtered unexpected plans is much less. Meanwhile, the segment model in Eraser concentrates more on balancing the regression elimination and impact on the improvement, so it only filters a small number of queries (such as the left part of Figure 6(b)) to avoid affecting the overall performance. This implies the adaptiveness of Eraser, which could be tuned to make the best trade-off between regression risks and improvement benefits.

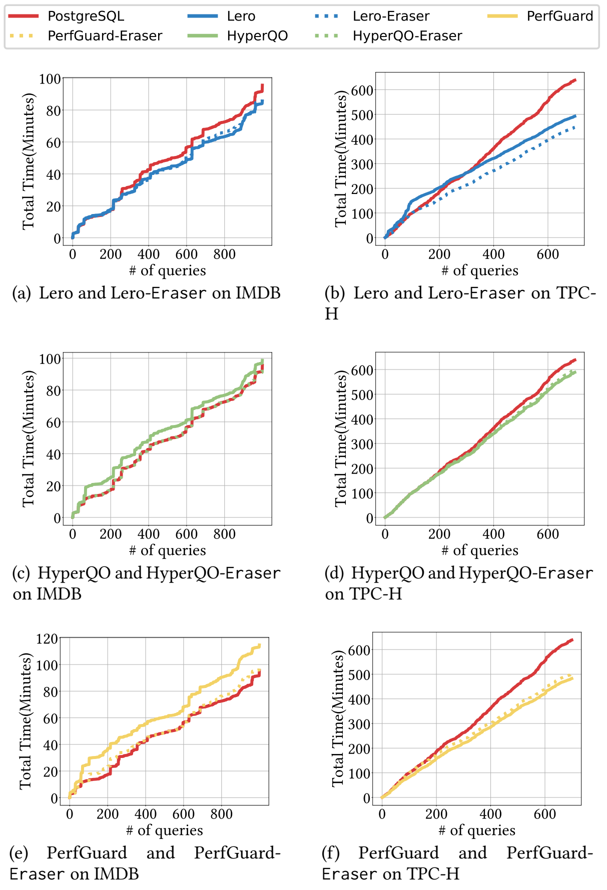
6.2.3 Regression Elimination by More Training Data. Another possible way to eliminate regression is to enlarge the training workload of learned query optimizer. For fair of comparison, we add the queries generated in unexpected plan explorer into the training workload and compare the performance of Lero with more training queries (denoted as Lero+). Their results of the stable models on IMDB and STATS benchmarks are shown in Figure 7.

We find that Lero+ still has a severe performance regression while Eraser could eliminate all regression. This is because a small volume of additional training data can not guarantee the risk models to generalize well on all unexpected plans. However, they are enough to qualitatively evaluate the performance of risk models for each coarse-grained subspace of unexpected plans, as we do in Eraser. Meanwhile, some regressions are caused due to the limited capacity of the risk model. This intrinsic under-fitting problem may not be resolved by only using more training data. Due to the same



(a) Lero and Lero-Eraser (b) HyperQO and HyperQO-Eraser (c) PerfGuard and PerfGuard-Eraser (d) Performance of Stable Models

Figure 8: Performance of learned query optimizer on dynamic data.



(a) Lero and Lero-Eraser on IMDB (b) Lero and Lero-Eraser on TPC-H

(c) HyperQO and HyperQO-Eraser on IMDB (d) HyperQO and HyperQO-Eraser on TPC-H

(e) PerfGuard and PerfGuard-Eraser on IMDB (f) PerfGuard and PerfGuard-Eraser on TPC-H

Figure 9: Performance curve of learned query optimizer since deployment.

reason, adding more training queries can not ensure a positive impact on the performance. For example, Lero+'s performance is worse than Lero's on STATS with more training data. A similar phenomenon is also observed for PerfGuard on the STATS benchmark (see the middle figure in Figure 5).

6.2.4 Performance Curve since Deployment. We show the performance curve since deployment in the online evaluation scenario. Figure 9 shows the performance of all learned query optimizers on IMDB and TPC-H. The results on STATS are similar. We make the following observations:

1) When the learn query optimizer consistently performs worse than PostgreSQL, e.g., HyperQO and PerfGuard on IMDB, Eraser could help to eliminate the regressions and attain comparable performance w.r.t. PostgreSQL. On the contrary, when the learn query optimizer consistently performs better than PostgreSQL, e.g., PerfGuard on TPC-H, Eraser makes very little impact on its performance. Once again, this verifies the effectiveness of Eraser.

2) In other cases where the learn query optimizer could perform better than PostgreSQL after only seeing enough training queries, e.g., Lero on TPC-H and IMDB. Eraser could still eliminate the regression at the very early stage and bring non-negative, even possible good (e.g. Lero on TPC-H), impact at the later stage. This once again verifies the adaptiveness of Eraser to risk models with different performance in different stages.

Meanwhile, we find that in both evaluation scenarios, Eraser could work well on different learned query optimizers with different plan exploration strategies and risk models. This is due to the models in Eraser are totally independent of the underlying systems, so they are applicable to any learned query optimizer. Whereas, the other methods, such as the ensemble method in HyperQO, can not be easily extended to other learned query optimizers.

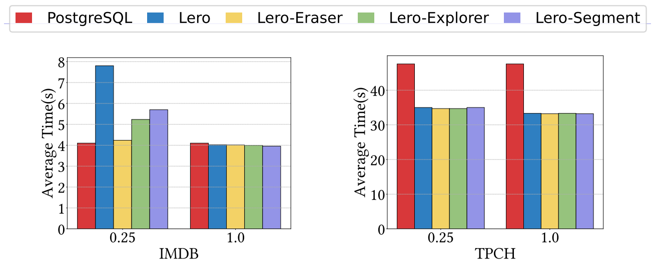
6.3 Performance on Dynamic Data

In this experiment, we evaluate the performance of Eraser in the setting of dynamic data. To this end, we insert 50% of the data into the database at the beginning and insert 12.5% of the data after receiving every 25% of training queries. Figure 8(a-c) shows the performance curve of several learned query optimizers since deployment on the training workload of STATS benchmark. Figure 8(d) illustrates the performance of the stable models on test workload. We observe that Eraser could still help to eliminate the regression with very little impact on the improvement. This is because both the unexpected plan explorer and segment model in Eraser work on the space of plan features, which are totally independent of the data distributions. As a result, Eraser is robust to data changes.

Meanwhile, Eraser could eliminate the regression at the very early stage. This is because the unexpected plan explorer in Eraser has been trained on the plan space using generated queries, so the initial performance of Eraser does not rely on training queries. In the later stage, the segment model in Eraser is gradually updated with the information of training queries. Therefore, Eraser could bring a non-negative, even possibly good, impact. This verifies the success of our design in Eraser.

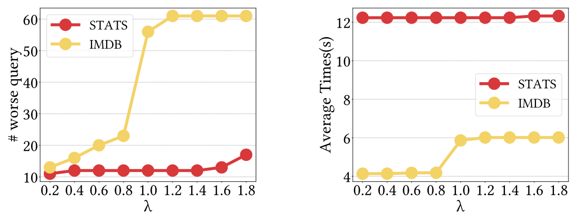
6.4 Ablation Analysis

We conduct an ablation analysis to investigate the effects of each component, namely unexpected plan explorer and segment model,



(a) Performance on IMDB (b) Performance on TPC-H

Figure 10: Ablation analysis for two components.



(a) # of Regression Queries (b) Average Execution Time

Figure 11: Effects of parameter λ.

in Eraser. We attach Lero with only each component and compare their performance. The results of stable models on test workloads of the IMDB and TPC-H benchmarks are shown in Figure 10. We have the following findings:

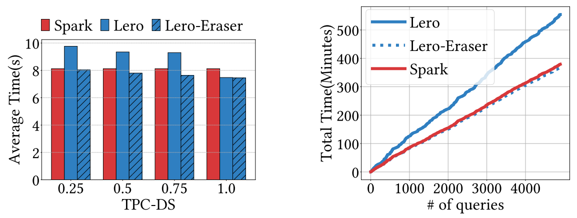
1) Each component could contribute to eliminating the performance regression, but the effect is worse than combining them together. For example, on IMDB benchmark with 25% training data, the unexpected plan explorer and segment model could eliminate 68% and 57% regression of Lero, respectively. However, by using both components, Lero-Eraser could eliminate 96% of the regression. This is because the two components filter plans with regressions caused by different reasons. Unexpected plan explorer eliminates plans with unseen features that the model can not generalize well while the segment model eliminates remaining plans with low learning accuracy.

2) Each component has little impact on the improvement of Lero. This is because the candidate plans with benefits are often not unexpected plans (see Figure 2(a)), which would often not be filtered by the unexpected plan explorer. Meanwhile, the learning difficulty of the segment model is much smaller than the risk model. Thus, it is easier to attain our desired goal in loss function to balance the regressions and benefits.

We also observe the similar phenomenon on other learned query optimizers with Eraser. We omit the details due to space limitation. This indicates the effectiveness of the two-stage filtering strategy in Eraser.

6.5 Effects of Parameter λ

We study the effects of λ on balancing the elimination of regression and the impact on improvement. We vary λ from 0.2 to 1.8 on IMDB and STATS benchmarks. Figure 11 illustrate the number of regression queries (in the left part) and average execution time (in the right part) of the stable models in Lero-Eraser on the test workloads. We find that:



(a) Performance of Stable Models (b) Online Performance

Figure 12: Performance of learned query optimizer on Spark.

1) By increasing λ, the number of regression queries also increases. This is simply because a larger λ encourages to improve the benefits so the segment model reserve more candidate plans.

2) For different λ, the execution time may vary. By filtering different plans, the total benefit $\mathcal{B}'$ and the total regression $\mathcal{R}'$ varies, so the time variance, i.e., $\mathcal{B}' - \mathcal{R}'$ may be different. On different datasets, the volume of $\mathcal{B}'$ and $\mathcal{R}'$ differs, so the varying speed is also different. In our dataset, the execution time increases on IMDB but tends to keep stable on STATS.

6.6 Performance on Spark

In this set of experiments, we test the performance of Eraser on distributed database. We deploy Eraser on Lero on Spark and apply the TPC-DS benchmark for evaluation. Figure 12 shows the execution time of Lero and Lero-Eraser in the two evaluation scenarios. We find that, for the stable models of Lero (in Figure 12(a)), it only performs better than the original query optimizer in Spark when trained on 100% of data. In the online evaluation scenario (in Figure 12(b)), Lero consistently performs worse than Spark. Fortunately, Eraser could help to eliminate regressions of Lero in all cases to attain comparable, or slightly better, performance of Spark. This indicates the generality on the design choices of our Eraser, which is applicable to different DBMSes with different query optimization mechanisms.

7 CONCLUSIONS AND FUTURE WORK

Performance regression commonly occurs in learned query optimizers and has a serious impact on their applicability and stability. In this paper, we design a system called Eraser to resolve this challenging problem. Eraser could be deployed on top of any existing learned query optimizer to eliminate the performance regression while preserving the performance improvement. Eraser adopts a two-stage strategy to identify the estimation accuracy of each plan, where the first stage qualitatively filters all unpromising plans with high risks and the second stage quantitatively evaluates the estimation quality of remaining plans. The final plan is selected to make the best trade-off between benefit and risk. Extensive experiments on different learned query optimizers in PostgreSQL and Spark exhibit the effectiveness and generality of Eraser.

In future work, we try to internally integrate Eraser into the plan exploration strategy and prediction models in learned query optimizers. Meanwhile, we consider extending Eraser to other tasks in learned databases, such as knob tuning [18, 24, 36], index recommendation [4, 7, 25, 31–33] and view advisor [8, 16, 49].

REFERENCES

[1] 2022. HyperQO Implementation. https://github.com/yxfish13/HyperQO.

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

[3] Omer Achrack, Raizy Kellerman, and Ouriel Barzilay. 2020. Multi-loss sub-ensembles for accurate classification with uncertainty estimation. *arXiv preprint arXiv:2010.01917* (2020).

[4] Surajit Chaudhuri and Vivek R Narasayya. 1997. An efficient, cost-driven index selection tool for Microsoft SQL server. In *VLDB*, Vol. 97. San Francisco, 146–155.

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

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

[7] Bailu Ding, Sudipto Das, Ryan Marcus, Wentao Wu, Surajit Chaudhuri, and Vivek R. Narasayya. 2019. AI Meets AI: Leveraging Query Executions to Improve Index Recommendations. In *SIGMOD Conference*. ACM, 1241–1258.

[8] Tansel Dokeroglu, Murat Ali Bayir, and Ahmet Cosar. 2015. Robust heuristic algorithms for exploiting the common tasks of relational cloud database queries. *Applied Soft Computing* 30 (2015), 72–82.

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

[10] 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.

[11] Yuxing Han, Ziniu Wu, Peizhi Wu, Rong Zhu, Jingyi Yang, Liang Wei Tan, Kai Zeng, Gao Cong, Yanzhao Qin, Andreas Pfadler, et al. 2021. Cardinality estimation in dbms: A comprehensive benchmark evaluation. *arXiv preprint arXiv:2109.05877* (2021).

[12] 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.

[13] H. M. Sajjad Hossain, Marc T. Friedman, Hiren Patel, Shi Qiao, Soundar Srinivasan, Markus Weimer, Remmelt Ammerlaan, Lucas Rosenblatt, Gilbert Antonius, Peter Orenberg, Vijay Ramani, Abhishek Roy, Irene Shaffer, and Alekh Jindal. 2021. PerfGuard: Deploying ML-for-Systems without Performance Regressions, Almost! *Proc. VLDB Endow.* 14, 13 (2021), 3362–3375.

[14] Gao Huang, Yixuan Li, Geoff Pleiss, Zhuang Liu, John E Hopcroft, and Kilian Q Weinberger. 2017. Snapshot ensembles: Train 1, get m for free. *arXiv preprint arXiv:1704.00109* (2017).

[15] Harish D Pooja N Darera Jayant and R Haritsa. 2008. Identifying robust plans through plan diagram reduction. In *VLDB*, Vol. 24. Citeseer, 25.

[16] Alekh Jindal, Konstantinos Karanasos, Sriram Rao, and Hiren Patel. 2018. Selecting subexpressions to materialize at datacenter scale. *Proceedings of the VLDB Endowment* 11, 7 (2018), 800–812.

[17] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter Boncz, and Alfons Kemper. 2018. Learned cardinalities: Estimating correlated joins with deep learning. *arXiv preprint arXiv:1809.00677* (2018).

[18] Mayuresh Kunjir and Shivnath Babu. 2020. Black or white? how to develop an autotuner for memory-based analytics. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1667–1683.

[19] Meghdad Kurmanji and Peter Triantafillou. 2022. Detect, Distill and Update: Learned DB Systems Facing Out of Distribution Data. *CoRR* abs/2210.05508 (2022).

[20] Balaji Lakshminarayanan, Alexander Pritzel, and Charles Blundell. 2017. Simple and scalable predictive uncertainty estimation using deep ensembles. *Advances in neural information processing systems* 30 (2017).

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

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

[23] Beibin Li, Yao Lu, and Srikanth Kandula. 2022. Warper: Efficiently Adapting Learned Cardinality Estimators to Data and Workload Drifts. In *SIGMOD Conference*. ACM, 1920–1933.

[24] Guoliang Li, Xuanhe Zhou, Shifu Li, and Bo Gao. 2019. Qtune: A query-aware database tuning system with deep reinforcement learning. *Proceedings of the VLDB Endowment* 12, 12 (2019), 2118–2130.

[25] Martin Luhring, Kai-Uwe Sattler, Karsten Schmidt, and Eike Schallehn. 2007. Autonomous management of soft indexes. In *2007 IEEE 23rd International Conference on Data Engineering Workshop*. IEEE, 450–458.

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

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

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

[29] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional Neural Networks over Tree Structures for Programming Language Processing. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA*, Dale Schuurmans and Michael P. Wellman (Eds.). AAAI Press, 1287–1293. http://www.aaai.org/ocs/index.php/AAAI/AAAI16/paper/view/11775

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

[31] Wendel Góes Pedrozo, Júlio Cesar Nievola, and Deborah Carvalho Ribeiro. 2018. An adaptive approach for index tuning with learning classifier systems on hybrid storage environments. In *Hybrid Artificial Intelligent Systems: 13th International Conference, HAIS 2018, Oviedo, Spain, June 20-22, 2018, Proceedings 13*. Springer, 716–729.

[32] Zahra Sadri, Le Gruenwald, and Eleazar Leal. 2020. Online Index Selection Using Deep Reinforcement Learning for a Cluster Database. In *ICDE Workshops*. IEEE, 158–161.

[33] Karl Schnaitter, Serge Abiteboul, Tova Milo, and Neoklis Polyzotis. 2007. On-line index selection for shifting workloads. In *2007 IEEE 23rd International Conference on Data Engineering Workshop*. IEEE, 459–468.

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

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

[36] Jian Tan, Tieying Zhang, Feifei Li, Jie Chen, Qixing Zheng, Ping Zhang, Honglin Qiao, Yue Shi, Wei Cao, and Rui Zhang. 2019. ibtune: Individualized buffer tuning for large-scale cloud databases. *Proceedings of the VLDB Endowment* 12, 10 (2019), 1221–1234.

[37] Kostas Tzoumas, Amol Deshpande, and Christian S Jensen. 2011. Lightweight graphical models for selectivity estimation without independence assumptions. *Proceedings of the VLDB Endowment* 4, 11 (2011), 852–863.

[38] Matias Valdenegro-Toro. 2019. Deep sub-ensembles for fast uncertainty estimation in image classification. *arXiv preprint arXiv:1910.08168* (2019).

[39] Laurens Van der Maaten and Geoffrey Hinton. 2008. Visualizing data using t-SNE. *Journal of machine learning research* 9, 11 (2008).

[40] 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.

[41] Yeming Wen, Dustin Tran, and Jimmy Ba. 2020. Batchensemble: an alternative approach to efficient ensemble and lifelong learning. *arXiv preprint arXiv:2002.06715* (2020).

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

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

[44] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, Ion Stoica, et al. 2010. Spark: Cluster computing with working sets. *HotCloud* 10, 10-10 (2010), 95.

[45] Erkang Zhu, Dong Deng, Fatemeh Nargesian, and Renée J. Miller. 2019. JOSIE: Overlap Set Similarity Search for Finding Joinable Tables in Data Lakes. In *SIGMOD Conference*. ACM, 847–864.

[46] Rong Zhu, Wei Chen, Bolin Ding, Xingguang Chen, Andreas Pfadler, Ziniu Wu, and Jingren Zhou. 2022. Lero: A Learning-to-Rank Query Optimizer. *Proc. VLDB Endow.* 16, 6 (2022), 1466–1479.

[47] 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.

[48] 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.

[49] Daniel C Zilio, Calisto Zuzarte, Sam Lightstone, Wenbin Ma, Guy M Lohman, Roberta J Cochrane, Hamid Pirahesh, Latha Colby, Jarek Gryz, Eric Alton, et al. 2004. Recommending materialized views and indexes with the IBM DB2 design advisor. In *International Conference on Autonomic Computing, 2004. Proceedings.* IEEE, 180–187.