Eraser: Eliminating Performance Regression on Learned Query Optimizer

Lianggui Weng[#]
Alibaba Group
Hangzhou, China
lianggui.wlg@alibaba-inc.com

Rong Zhu[#,*]
Alibaba Group
Hangzhou, China
red.zr@alibaba-inc.com

Di Wu
Alibaba Group, HUST
Hangzhou, China
woodybryant.wd@alibaba-inc.com

Bolin Ding[*]
Alibaba Group
Hangzhou, China
bolin.ding@alibaba-inc.com

Bolong Zheng[*]
HUST
Wuhan, China
bolongzheng@hust.edu.cn

Jingren Zhou[*]
Alibaba Group
Hangzhou, China
jingren.zhou@alibaba-inc.com

ABSTRACT

Efficient query optimization is crucial for database management systems. Recently, machine learning models have been 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 feature values that are seen for the first time. The second stage clusters plans in a more fine-grained manner and evaluates each cluster according to the prediction quality of learned query optimizers for selecting the final execution plan. Eraser can 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 bringing 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.

PVLDB Reference Format:
Lianggui Weng, Rong Zhu, Di Wu, Bolin Ding, Bolong Zheng Jingren Zhou. Eraser: Eliminating Performance Regression on Learned Query Optimizer. PVLDB, 17(5): XXX-XXX, 2024.
doi:XX.XX/XXX.XX

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

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 a cost-based model that estimates the cost of execution plans based on simple statistics and experience-driven rules [32]. However, the estimated costs are often shown to have large errors [11, 14, 20, 35], 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 [5, 24, 25, 27, 41, 42, 45]. These optimizers apply machine learning (ML) 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. Second, all candidate plans $P \in \mathcal{P}_Q$ are fed into an ML model to predict its cost $C(P)$. 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, 24, 45], 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 [5, 24, 25, 27, 41, 42, 45].

The learned models can not accurately predict the exact cost of some data due to numerous reasons, such as the inherent difficulty of the learning problem [20, 26, 45], the low generalization ability of the prediction model on new data, the under-fitting on training data due to insufficient training data, loss of features, noisy labels and inappropriate model training methods. Due to such intrinsic drawbacks, the performance regressions of learned query optimizers seem inevitable, no matter how we improve the models in learned query optimizers. This is very harmful, or even unacceptable, to database systems which require high stability.

[#] Equal Contribution. [*] Corresponding authors.

Proceedings of the VLDB Endowment, Vol. 17, No. 5 doi:XX.XX/XXX.XX

In the literature, there exists very little work on eliminating the performance regression, especially on learned query optimizer [46]. [21] and [18] propose methods to enhance the robustness in dynamic settings by updating models in the proper time. They are post-processing methods but do not detect and eliminate regression before query execution. [42] tries to reduce the regression using the ensemble methods [4, 13, 19, 36, 39], but it is time-consuming 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 benefits and risks 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 can be deployed as an external plugin on top of any existing learned query optimizer. Eraser can be tuned to eliminate its performance regression while bringing 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 can filter out all highly risky candidate plans but reserve those with high prediction accuracy for plan selection. However, learning the exact prediction accuracy is very challenging, which is as difficult as learning the accurate cost of each plan [11, 14, 20, 26, 35]. In Eraser, we try to simplify the learning tasks while still preserving enough knowledge for plan identification.

Specifically, Eraser adopts 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 subspaces, each with one or more unseen feature values. Then, we generate plans in a small number of representative subspaces. Based on the model evaluation results on these plans, we can classify all subspaces 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 into 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 of benefits. Both the unexpected plan explorer and the segment model are lightweight.

Through 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 can 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 influence 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 [5, 42, 45] and different DBMSes, i.e., PostgreSQL and Spark [43].

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 the learned query optimizer.

2) We design Eraser, a system that can 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.

2 PRELIMINARIES

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.

Recently, a number of learned query optimizers [5, 24, 25, 41, 42, 45] are proposed to provide instance-level query optimization. Their procedures can 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 $\mathcal{P}_Q = \{P_0, P_1, \dots, P_k\}$ 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 can 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, including Neo [25], Balsa [41], Bao [24], HyperQO [42], Lero [45], PerfGuard [5] and some other works [12, 16, 26, 28, 33, 47], apply different plan exploration strategies and risk models, but they can all be subsumed under this framework. Due to space limits, we defer the details to Appendix A in the full version [40].

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 suffer a heavy performance regression on some queries due to numerous reasons: 1) the candidate set $\mathcal{P}_Q$ does not contain plans better than P_b; 2) the risk model can not generalize well on new data/workload, especially in dynamic settings; and 3) 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 $\mathcal{Q}$ be a workload where each query $Q \in \mathcal{Q}$ occurs with the probability $\Pr_Q$. Formally, a learned query optimizer learned_opt in our framework generates an execution plan P_r for a query $Q \in \mathcal{Q}$

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

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 fewer 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 perf_elim can be interpreted as a plugin function (in any learned optimizer learned_opt) which filters out unreliable candidates and selects a different plan P'_r

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

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 perf_elim that is able to eliminate regressions but brings little impact on the benefits. Formally, our problem is stated as follows:

Performance Regression Elimination Problem

Input: a learned query optimizer 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 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 small value of λ, e.g., $\lambda = 0.8$ in our experiments, emphasizes filtering out risky plans, even at the expense of removing some good plans. It can be applied when the learned risky models are not very accurate due to scarce training data or varied workloads. Whereas, a large value of λ encourages perf_elim to improve the benefit by reserving more plans. We recommend using it when the learned models are more accurate with sufficient training data and stable workloads. Due to space limits, we reserve the discussion on tuning hyper-parameter λ in Appendix B and the analysis of existing solutions for eliminating performance regressions in Appendix C in the full version [40].

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 prediction accuracy for each plan, so that we can filter out all highly risky plans and only reserve those with high prediction accuracy for plan selection. However, existing learned query optimizers often provide very little knowledge on the prediction accuracy of plans. Meanwhile, it is very difficult (or even impossible) to learn the exact prediction accuracy of each plan, which is as challenging as learning the accurate cost of each plan [24, 26, 42].

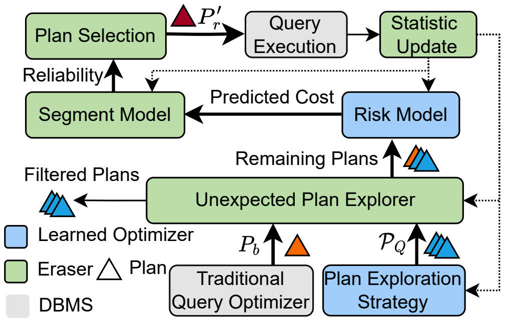
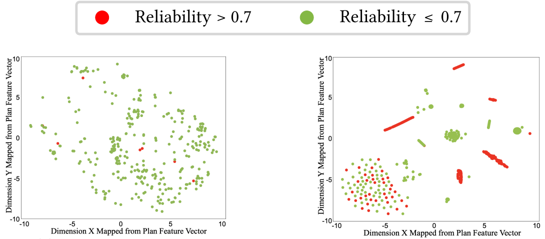


Figure 1: The system architecture and pipeline of Eraser.



(a) Unexpected Plans (b) Remaining Plans

Figure 2: Prediction quality (reliability) of the risk model in Lero on plans of the TPC-H benchmark. All plans in the high-dimensional feature space are mapped onto the two-dimensional space using the t-SNE method in [37].

To this end, we design other tasks that are much easier but still preserve enough information to identify truly good plans. We design a system, called Eraser, to eliminate regression of learned 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 prediction quality of risk models. We illustrate them by an example in Figure 2. Specifically, we employ the t-SNE [37] method to map plans in the TPC-H benchmark [7] onto a two-dimensional space. Each dimension is a snapshot of some dimensions in the plan feature space. Then, we mark all plans whose reliability (a metric of the prediction quality, see details in Section 5.3) is larger than or no more than the threshold 0.7 (selected by our expert knowledge) in red and green color, respectively. Notice that, the phenomenon shown in Figure 2 arises from the intrinsic drawbacks of most ML models. Thus, it is also applicable to other learned query optimizers and benchmarks. 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 $\mathcal{P}_Q$ 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 subspaces 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 prediction quality of the risk model on these plans.

Based on the reliability interval, we could further filter some unpromising plans with low prediction 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 only reserve plans with high reliability, which may miss some improvement opportunities but could reduce the regression to a very low level. 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 subspaces according to the domain of features. Then, we select a small number of unexpected plan subspaces and generate some plans in each subspace. Based on these plans, we 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 consider the SQL query Q in the following form:

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

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 can 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 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.

Feature Selection. In the literature work [24, 25, 41, 42, 45], the query plans are featured as vectors and fed into the risk model for cost prediction. We conceptually divide frequently-used features into two types: the plan-level features and the data-level features. The plan-level features, such as the join relations, filtering predicates and operator types, affect the execution behaviors of plans on the database execution engine. The data-level features, such as the estimated cardinality and data distribution, provide additional statistical information to measure the execution cost. The learned query optimizers require sufficient information to predict the plan execution time. Therefore, they apply both plan-level and data-level features to characterize the execution behavior and cost of plans.

However, the goal of `Eraser` is different. It aims at clustering the plans and then evaluating the prediction accuracy of the learned models on each cluster of plans. (refer to details in Section 4.2 and Section 5). Thus, the plan-level features are more important as they are more comprehensive to distinguish different plans in a macro-view. Since `Eraser` does not require predicting the execution cost, the data-level features are not very necessary. By our expert knowledge, we consider the following widely used features:

1) *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 the join type is {MJ, HJ}. Notice that, we only consider the scan type and can support the scan operator on any number of columns.

2) *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 is provided by users or found by an auto-exploration method proposed in [44]. We use a binary variable to encode the existence of each join relation in the vector.

3) *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 can be equivalently converted into this form as

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

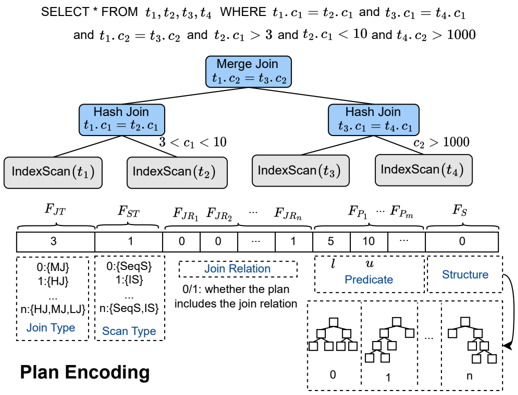

SELECT * FROM t_1, t_2, t_3, t_4 WHERE $t_1.c_1 = t_2.c_1$ and $t_3.c_1 = t_4.c_1$
and $t_1.c_2 = t_3.c_2$ and $t_2.c_1 > 3$ and $t_2.c_1 < 10$ and $t_4.c_2 > 1000$
Merge Join $t_1.c_2 = t_3.c_2$
Hash Join $t_1.c_1 = t_2.c_1$
Hash Join $t_3.c_1 = t_4.c_1$
$3 < c_1 < 10$
$c_2 > 1000$
IndexScan(t_1)
IndexScan(t_2)
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.

In our implementation, we divide the domain of each continuous attribute to a number of small intervals, so ϵ can be set to a value smaller than the range of each divided interval.

4) *structure*: a categorical variable indicating the shape of the plan tree. Each value corresponds to a specific form of the plan ignoring the physical 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.

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 to enhance the capability of `Eraser`. Specifically, for each new plan-level or data-level feature, we model it as a categorical or continuous variable. Then, we process all categorical features, such as GROUP BY and ORDER BY operators, in a similar way to the join type feature. For all features with continuous values, we process them in a similar way to the filtering predicate. It is important to highlight that the features of `Eraser` are independent of the learned query optimizer. Thus, even if certain features cannot be covered by `Eraser`, it still has the capability to effectively eliminate regressions.

The effectiveness of our selected features is verified by the experimental results in Section 6. We also note that the impact of each plan-level feature may be different for `Eraser`. It is worth investigating which subset of features plays the most important role in distinguishing plans. However, this topic falls within the realm of feature engineering and goes beyond the scope of this paper. We reserve it as an important direction in our future work.

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 subspaces and generates plans in some representative subspaces 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. 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 as the unexpected plan space contains $|\mathcal{U}| = \prod_i |D_i| - \prod_i |S_i|$ points. For continuous features such as the filtering predicate, their domains are divided into a number of small intervals, and $|D_i|$ refers to the number of divided intervals. 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:

1) 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.

2) The risk model may have similar prediction accuracy for nearby value $d_i, d'_i \in D_i$ of the continuous feature F_i. This is because these plans may have similar actual values and estimated values. This implies that we can 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. Notably, our division method splits each feature independently. 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 (line 5 in the Algorithm). In general, we apply three methods for division (line 10 in the Procedure). 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 into two subsets U_i^1 and U_i^2. For domain U_i with 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. We can control the splitting granularity to balance the evaluation efficiency and accuracy.

The Recur_Split process terminates if $|U_i^j|$ is smaller than a user-specified threshold (line 1). 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 (lines 2–3). 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 (line 5 by our first observation). Otherwise, we mark all points p in the evaluated subspace as precise (line 7).

Note that, for each U_i^j, we examine all points in the subspace $S_1 \times \cdots \times S_{i-1} \times U_i^j \times S_{i+1} \times \cdots \times S_n$ altogether. Since each U_i^j

contains at least one element in U_i, the number of the subspaces is at most $\sum_i |U_i|$, which is much less than the number of all points ($\prod_i |D_i| - \prod_i |S_i|$) in the space. Meanwhile, 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 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).

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 popular DBMS such as PostgreSQL, MySQL and SQL Server. We can 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. Similarly, 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 into DBMS according to the join orders.
- Fourth, 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 can be either a pointwise regression model [24, 42] or a pairwise classification model [5, 45]. We process it in different ways.

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

Procedure Recur_Split($F_i, U_i, D_1, \dots, D_n, S_1, \dots, S_n$)
if $|U_i|$ is smaller enough according to feature $F_i$ then
    generate plans falling into $S_1 \times \cdots \times S_{i-1} \times U_i^j \times S_{i+1} \times \cdots \times S_n$
    evaluate model performance using generated plans
    if the model perform bad then
        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
    else
        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
    end if
else
    divide $U_i$ into disjoint subset $U_i^1, U_i^2, \dots, U_i^k$
    for each $U_i^j$ where $1 \le j \le k$ do
        Recur_Split($F_i, U_i^j, D_1, \dots, D_n, S_1, \dots, S_n$)
    end for
end if
```

If the risk model is pointwise, it takes a plan P as input and predicts its estimated time $\widehat{C}(P)$ to approximate the exact time $C(P)$. We define the accuracy metric $e(P)$ of the plan P as

$$e(P) = \min(|\frac{\widehat{C}(P)}{C(P)} - 1|, \text{ UB}) \tag{3}$$

Here, the 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 by a grid search method in our experiments. We compute the average ratio $\overline{e}(P)$ of all unexpected plans generated for the subspace. We always have $\overline{e}(P) \in [0, \text{UB}]$. 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 can filter out more risky plans.

When the risk model is pairwise, it takes a pair of plans P, P' and outputs a binary label to indicate which plan is better. For such models, we directly apply the pairwise accuracy w.r.t. the relative order to measure the model performance. Specifically, for each plan P in the subspace, we consider all other plans P' and collect pairs of plans (P, P'). We then compute $e(P)$ as the proportion of pairs where the risk model accurately finds the better plan. We mark the subspace using a threshold α on the average value $e(P)$ in the same way.

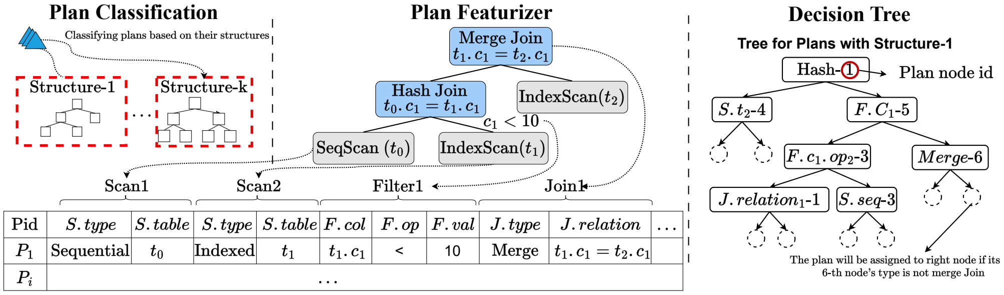
5 SEGMENT MODEL AND PLAN SELECTION

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 subspaces


Plan Classification
Classifying plans based on their structures
Structure-1
Structure-k
Plan Featurizer
Merge Join $t_1.c_1 = t_2.c_1$
Hash Join $t_0.c_1 = t_1.c_1$
IndexScan(t_2)
$c_1 < 10$
SeqScan (t_0)
IndexScan(t_1)
Decision Tree
Tree for Plans with Structure-1
Hash-1
Plan node id
$S.t_2$-4
$F.C_1$-5
$F.c_1.op_2$-3
$Merge$-6
$J.relation_1$-1
$S.seq$-3
The plan will be assigned to right node if its 6-th node's type is not merge Join

	Scan1		Scan2		Filter1			Join1		
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 classification process based on their structure. The middle part is the plan featurizer. The right part is a forest, where each tree represents the clustering model for a specific structure.

$\mathcal{G} = \{G_1, G_2, \ldots, G_k\}$. 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, our goal is, after clustering the remaining plans, to 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). 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 significantly updated. Second, the behavior of the segment model should be deterministic and better to be explainable. This is because the behavior of the deep model is often uncertain [38]. 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.

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 makes it difficult to be encoded as a vector having a fixed length. To address this problem, existing deep models [24, 25, 45] encode the feature of each node and then generate the tree-level encoding using a convolution operation [27]. 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 a bit-wise sum or mean value [8]. 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. This structure is defined as the simplified logical plan tree, which consists of logical operators (e.g., join and scan), while ignoring the specific physical operator type (e.g., hash join or seq scan), filtering predicates, and other details. It is important to note that "ignoring" is only applied to define the plan structure. We will apply these features to build our segment model.

Figure 4 illustrates an example of our segment model, where we list several possible plan structures in the left part. The feature vector of each plan shown in the middle part is obtained by concatenating the encoding vector on all nodes. Specifically, for all inner nodes 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. For each type of tree structure, we build the decision tree model using the encoded plans as shown in the right part.

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_Construction. First, we collect all plans $\mathcal{P}_N \subseteq \mathcal{P}_S$ with this kind of structure (line 2) and maintain it in the root node (line 3). Then, the algorithm calls the Procedure Model_Train() to build the decision tree in a recursive manner (line 4).

Specifically, in the Procedure Model_Train(), 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 each possible $f_{i,j}$, we split $\mathcal{P}_N$ into $\mathcal{P}_L$ and $\mathcal{P}_R$ on the left and right child according to $f_{i,j}$ and compute the loss (lines 2–3). That is, we select another execution plan P_r' to replace the original execution plan P_r using the plan selection method (discussed in Section 5.3) based on this tree structure. Then, we 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) (line 5). It is applied to split the set of plans $\mathcal{P}_N$ (lines 6–7). If f_i is a categorical variable, $\mathcal{P}_L$

```
Algorithm Model_Construction($\mathcal{P}_S$)
1: for each kind of structure do
2:     collect $\mathcal{P}_N \subseteq \mathcal{P}_S$ with this kind of structure
3:     set $\mathcal{P}_N$ to be the root node of the decision tree
4:     Model_Train($f_1, f_2, \ldots, f_t, \mathcal{P}_N$)
5: end for
```

```
Procedure 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
```

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 $\mathcal{P}_R = \mathcal{P}_N - \mathcal{P}_L$. After that, the procedure works in a recursive manner. It splits all plans in the parent node into two children each time using the best splitting feature value (lines 11 and 16). 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 (lines 8 and 13), typically, 5% of the training data size.

Remarkably, the three features, i.e., column, operator and filtering predicate, 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.

5.3 Plan Selection Method

Based on the segment tree model, we introduce how to select a final execution plan in this subsection. According to how the risk model is designed in existing learned query optimizer [5, 24, 25, 41, 42, 45], 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 a model for such a structure, it implies we have very limited knowledge of 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. We prefer to utilize $r(P)$ to reflect the error ratio between estimated and actual cost. Thus, we directly define $r(P) = \min(\frac{\widehat{C}(P)}{C(P)}, \text{UB})$. The UB serves a similar purpose as in $e(P)$. 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, the 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 $\overline{r}(\mathcal{P}_L)$ be the average reliability value of plans in $\mathcal{P}_L$. We correct the predicted cost $C(P)$ to $C'(P) = C(P)/\overline{r}(P)$. 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 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 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 pairs 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 can accurately find the better plan. $r(\mathcal{P}_L, \mathcal{P}'_L)$ can also indicate the confidence of the risk model on these plans. Similarly, we trust the risk model if $r(P, P')$ 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. Specifically, our experiments aim to answer the most crucial questions as follows:

- When Eraser is deployed on top of the existing learned query optimizer, how much performance regression can it eliminate, and how much impact it may cause to the benefit? (in Section 6.2)
- Can 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 parameters λ, α and β? (in Section 6.5 and Section 6.6)

6.1 Experimental Setup

Baselines. We use three representative learned query optimizers in our experiments. Specifically, **HyperQO** [42] and **Lero** [45] 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 [24]. Besides, Lero is shown to perform better than Balsa [41]. Therefore, we do not apply Bao, Balsa and Neo [25] (which performs even worse than Bao) in our experiments. **PerfGuard** [5] is a learned query optimizer that supports any plan generation strategy. We also use Lero's plan exploration strategy to generate plans for PerfGuard. We implement HyperQO, Lero and PerfGuard using source code in [1], [2], and [3], respectively. Notably, these learned query optimizers are primarily designed for **PostgreSQL**. Therefore, we deploy all of them on the native query optimizer of PostgreSQL. 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 [24, 25, 42, 45]. We summarize their statistical information in Appendix D [40] due to space limitations. 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 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 a distributed database Spark. The remaining ones are used for experiments on PostgreSQL (in Section 6.2 to Section 6.5). In addition, to demonstrate the effectiveness of Eraser for eliminating performance regression in a more fine-grained manner. We have introduced two new test sets, FAST-IMDB and SLOW-IMDB, which consist of the top 30% and last 30% SQL queries, respectively, obtained by sorting the full test set based on the execution time of plans generated by PostgreSQL.

Evaluation Methods and Metrics. Following [24, 45], 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 learned query optimizer processes each training query online one by one and retrains its model using all observed queries after seeing every 100 queries. In our experiments, we report the average end-to-end (e2e) plan execution time of each query, which includes the plan generation and selection time of the original learned query optimizer, the plan examination and selection time of Eraser and the physical plan execution time.

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.

6.2 Performance of Eraser

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 can 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 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 the learned query optimizer. Meanwhile, the ensemble method in HyperQO can not ensure to eliminate regression. The reasons are analyzed in Appendix B in the full version [40]

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 can help to improve its performance to be comparable or slightly better than PostgreSQL. This indicates that Eraser can filter most of the unpromising plans that 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 can learn enough information to match our desired goal.

6.2.2 Extra Cost of Eraser. Table 1 shows the extra cost of Eraser and learned query optimizers. Specifically, the extra costs incurred by learned query optimizers include the time to generate candidate plans and select the best one using risk models. The extra costs incurred by Eraser include the time to examine and select plans using unexpected plan explorer and segment model components. We note that Eraser only consumes very little extra time, i.e., 0.17*ms* to 24*ms* per query. In comparison to the end-to-end plan execution time, the percentage is less than 0.001% (on TPC-H) to 0.5% (on IMDB), which is negligible. The model training time and training data collection time of Eraser are much smaller than the learned query optimizers. The training data collection time of Eraser is used to collect the execution time of the generated plans for the unexpected plan explorer component. It is important to note that this data collection is only performed once for each benchmark and is independent of the learned query optimizers. Meanwhile, the training data collection and model training are performed offline in the background using idle resources. This makes little impact on the online query performance. The memory cost of Eraser is less than 0.3MB, which is totally affordable for the DBMS. Therefore, the time and space cost to apply Eraser is very small.

6.2.3 Performance Analysis for Fast and Slow Queries. We demonstrate the effectiveness of Eraser for eliminating performance regression in a more fine-grained manner by evaluating the performance of these optimizers on the FAST-IMDB and SLOW-IMDB test sets.

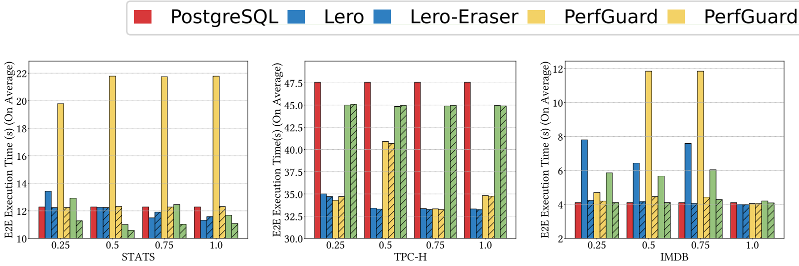


Figure 5: Performance of learned query optimizer with stable models.

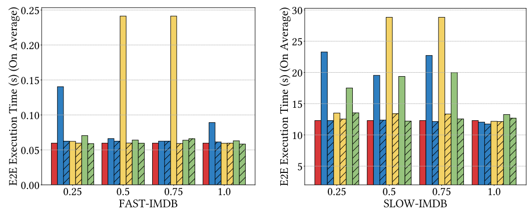
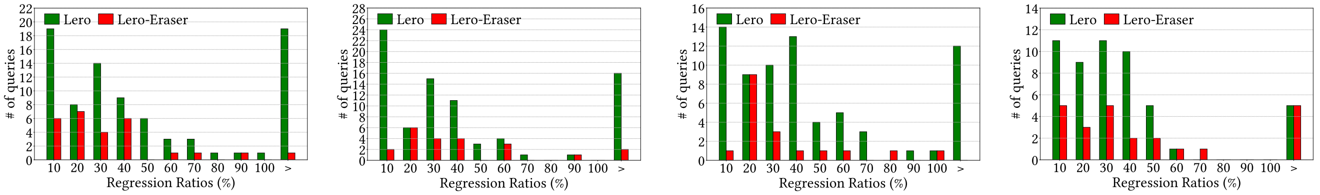


Figure 6: Performance on fast and slow queries.



(a) IMDB with 25% training data

(b) IMDB with 50% training data

(c) IMDB with 75% training data

(d) IMDB with 100% training data

Figure 7: Performance of Lero-Eraser on queries with different levels of regression.

Table 1: The cost of Eraser and learned query optimizers. The columns labeled "w. Eraser" and "w.o. Eraser" indicate whether or not Eraser is deployed on top of a learned query optimizer. All risk models of learned query optimizers are trained on 25% training data.

Metric	Learned Query Optimizer	IMDB		STATS		TPC-H	
		w. Eraser	w.o. Eraser	w. Eraser	w.o. Eraser	w. Eraser	w.o. Eraser
E2E Execution Time (ms) (On Average)	Lero	4,237	7,802	12,228	13,421	3,469	3,500
	HyperQO	4,101	5,858	11,271	12,917	4,507	4,500
	Perfguard	4,198	4,699	12,240	19,783	3,460	3,428

Metric	Learned Query Optimizer	IMDB		STATS		TPC-H	
		Eraser	Learned QO	Eraser	Learned QO	Eraser	Learned QO
Extra Time Cost (ms) (On Average)	Lero	24	201	4.5	28.8	4.1	16.5
	HyperQO	2.4	34.2	0.9	12.2	0.17	21.9
	Perfguard	2.0	176.8	15.3	37.9	11.1	27.3
Training Time (s)	Lero	14.8	199	5.9	118	3.46	132
	HyperQO	14.3	3157	3.45	1467	2.5	2550
	Perfguard	13.2	147	4.6	89	3.4	87
Data Collection Time (Min)	Lero	26	260	34	1,343	63	2,106
	HyperQO	26	733	34	300	63	685
	Perfguard	26	/	34	/	63	/
Model Size (MB)	Lero	0.27	1.35	0.13	1.22	0.02	1.23
	HyperQO	0.28	16.7	0.14	16.29	0.01	6.5
	Perfguard	0.27	16	0.08	1.3	0.02	1.3

Each learned query optimizer is trained with different proportions of the IMDB training data, and the results are presented in Figure 6. We have the following observations:

1) Performance regression often occurs when the learned query optimizer is not trained on the full 100% training set. This suggests that the risk model is easy to make imprecise predictions when the test and training sets are not from the same distribution. Consequently, an approach that can effectively eliminate performance regression is crucial.

2) Eraser demonstrates its ability to eliminate a significant portion of performance regression in both FAST-IMDB and SLOW-IMDB test sets. These results indicate that Eraser is not sensitive to the complexity of SQL queries and is suitable for a wide range of diverse SQL workloads.

By showcasing these fine-grained experiments, we provide strong evidence of the effectiveness of Eraser in mitigating performance regression and its suitability for optimizing various SQL queries.

6.2.4 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 that are trained on 25%, 50%, 75% and 100% of the IMDB training data are shown in Figure 7. Notice that the goal of Eraser is to balance the regression and the improvement to attain the best overall performance on all queries. Therefore, when models are trained using different volumes of data, Eraser will automatically adapt and exhibit varying behaviors:

1) In Figure 7 (a), (b) and (c), the model in the learned query optimizer is trained on the part of the training data. Thus, the model will encounter a number of plans having unseen feature values during the testing stage, and select a large number of plans with heavy regression. For such a scenario, the overall performance of the learned query optimizer is worse than the baseline PostgreSQL. At the time, Eraser will focus more on filtering these unexpected plans with high risks to ensure the learned query optimizer keeps comparable performance with PostgreSQL. The number of filtered plans is 57, 59 and 54 in Figure 7 (a), (b) and (c), respectively.

2) In Figure 7 (d), the model is trained on all data, and the overall performance of the learned query optimizer is better than the baseline. At this time, Eraser would be more concerned with balancing the regression elimination and impact on the improvement. Therefore, it only filters a small number of queries, i.e., only 28 in Figure 7 (d), to avoid affecting the overall performance. As a result, some query plans with large regression ratios may not be filtered by Eraser. For example, Eraser chooses to retain all plans that have a regression ratio exceeding 100%. Notably, in our Eraser method, we remove all plans in a subspace together (see technical details of the unexpected plan explorer in Section 4 and segment model in Section 5). Therefore, removing these plans with significant regressions may also result in the removal of other more plans in the same subspace with substantial performance benefits. As a

result, Eraser prefers to retain all plans in the subspace. Notably, although Eraser keeps more queries with large regression ratios, the overall performance of the learned query optimizer becomes better. This adaptiveness ensures Eraser to make the best trade-off to attain better overall performance.

6.2.5 Performance Curve since Deployment. We show the performance curve since deployment in the online evaluation scenario. Figure 8 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 learned query optimizer consistently performs worse than PostgreSQL, e.g., HyperQO and PerfGuard on IMDB, Eraser can help to eliminate the regressions and attain comparable performance w.r.t. PostgreSQL. On the contrary, when the learned 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 its effectiveness.

2) In other cases, the learned query optimizer can perform better than PostgreSQL after only seeing enough training queries, e.g., Lero on TPC-H and IMDB. Eraser can 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 performances in different stages.

Meanwhile, we find that in both evaluation scenarios, Eraser can work well on different learned query optimizers with different 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. Furthermore, the other methods, such as the ensemble method in HyperQO, can not effectively eliminate performance regression.

Due to space limits, we put more experiments in terms of Eraser into Appendix E in the full version [40]. First, we compare the performance of Eraser with another technical routine to eliminate regression by enlarging the training workload in Appendix E.1. Second, we investigate the effects of the splitting granularity in the unexpected plan explorer on the performance of Eraser in Appendix E.2. Third, we investigate the performance of Eraser on distributed databases, i.e., Spark 3.3 [43], in Appendix E.3.

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 9(a-c) shows the performance curve of several learned query optimizers since deployment on the training workload of the STATS benchmark. Figure 9(d) illustrates the performance of the stable models on test workload. We observe that Eraser can 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 can eliminate the regression at a 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 can 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, 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 can contribute to eliminating the performance regression, but the effect is worse than combining them. For example, on the IMDB benchmark with 25% training data, the unexpected plan explorer and segment model can eliminate 68% and 57% regression of Lero, respectively. However, by using both components, Lero-Eraser can 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 prediction 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's. Thus, it is easier to attain our desired goal in loss function to balance the regressions and benefits.

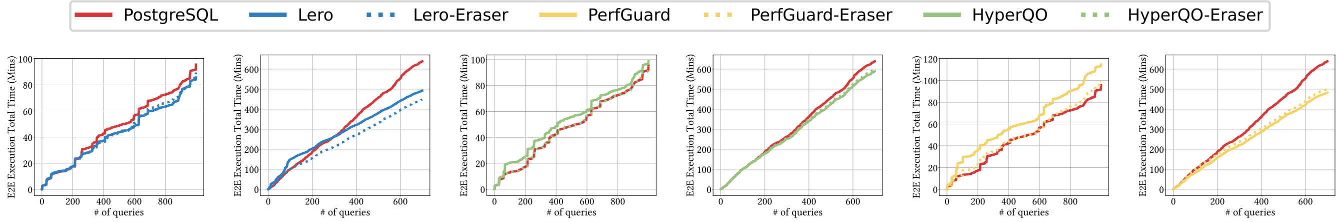
3) Figure 10(b) reports the ablation study results of Eraser on top-30% slowest queries. We find that the observations on slow queries are the same as other workloads. That is, the unexpected plan explorer eliminates plans with unseen features that the model can not generalize well, while the segment model eliminates remaining plans with low prediction accuracy. This once again verifies the effectiveness of Eraser on processing slow queries.

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 illustrates 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:

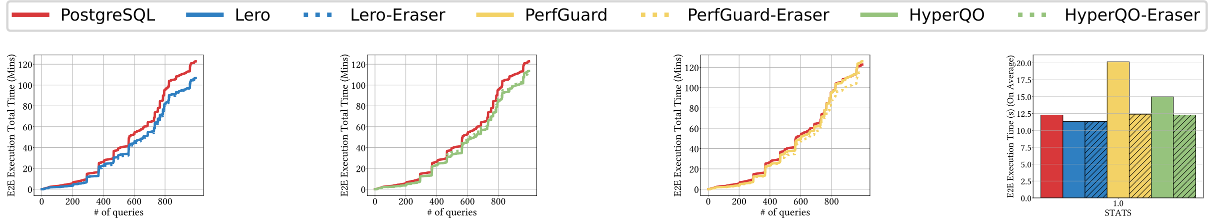
1) By increasing λ, the number of regression queries also increases. This is simply because a larger λ encourages to improve the benefits, so the segment model reserves more candidate plans.

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.



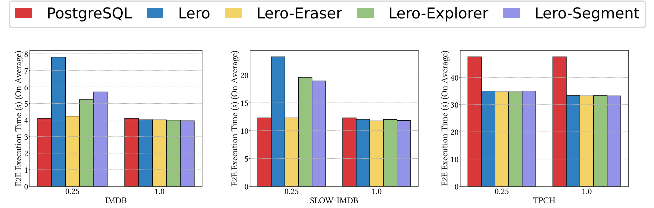
(a) Lero and Lero-Eraser on IMDB (b) Lero and Lero-Eraser on TPC-H (c) HyperQO and HyperQO-Eraser on IMDB (d) HyperQO and HyperQO-Eraser on TPC-H (e) PerfGuard and PerfGuard-Eraser on IMDB (f) PerfGuard and PerfGuard-Eraser on TPC-H

Figure 8: Performance curve of learned query optimizer since deployment.



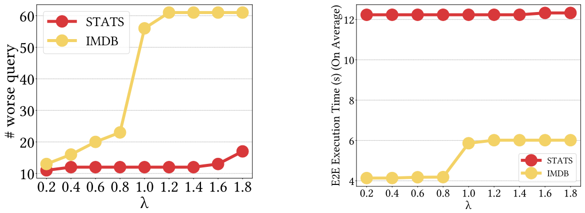
(a) Lero and Lero-Eraser (b) HyperQO and HyperQO-Eraser (c) PerfGuard and PerfGuard-Eraser (d) Performance of Stable Models

Figure 9: Performance of learned query optimizer on dynamic data.



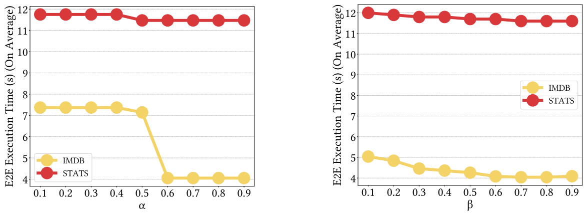
(a) Perf. on IMDB (b) Perf. on SLOW-IMDB (c) Perf. on TPC-H

Figure 10: Ablation analysis for two components.



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

Figure 11: Effects of parameter λ.



(a) Effect of parameter α (b) Effect of parameter β

Figure 12: Effects of parameter α and β.

6.6 Effects of Parameters β and α

We study the effects of α and β on balancing the elimination of regression and the impact on improvement. Specifically, we vary the parameter α or β from 0.1 to 0.9 and examine the average time of Lero on the IMDB and STATS benchmarks. Figure 12 illustrates the results. We have the following observations:

1) Increasing the value of α leads to an improvement in the overall performance. This is because the learned query optimizer tends to make imprecise predictions on unexpected plans. By pruning more subspaces of such unexpected plans, we can achieve a higher performance improvement. It is worth noting that the performance stabilizes after α reaches 0.7. At this time, the unexpected plan explorer has already filtered almost all risky plans.

2) Similarly, increasing the value of parameter β also results in a slight performance improvement. This phenomenon reflects the fact that the risk models exhibit poor performance on a subset of the data. By setting a reasonable value for β to filter out such risky plans, we can also eliminate some performance regressions.

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 can 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 prediction accuracy of each plan, where the first stage qualitatively filters all unpromising plans with high risks, and the second stage quantitatively evaluates the prediction quality of the 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 will 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 [17, 22, 34], index recommendation [6, 8, 23, 29–31] and view advisor [9, 15, 48].

Acknowledgements. This work was supported by grants from the National Natural Science Foundation of China (No. 62372194) and the program of Alibaba Innovative Research.

REFERENCES

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

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

[3] 2022. PerfGuard implementation. https://github.com/WoodyBryant/Perfguard.

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

[5] Remmelt Ammerlaan, Gilbert Antonius, Marc Friedman, HM Sajjad Hossain, Alekh Jindal, Peter Orenberg, Hiren Patel, Shi Qiao, Vijay Ramani, Lucas Rosenblatt, et al. 2021. PerfGuard: Deploying ML-for-systems without performance regressions, almost! *Proceedings of the VLDB Endowment* 14, 13 (2021), 3362–3375.

[6] 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.

[7] Transaction Processing Performance Council(TPC). 2021. TPC-H vesion 2 and version 3. http://www.tpc.org/tpch/.

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

[9] 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.

[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] 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.00109* (2017).

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

[15] 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.

[16] 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).

[17] 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.

[18] Meghdad Kurmanji and Peter Triantafillou. 2023. Detect, distill and update: Learned DB systems facing out of distribution data. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–27.

[19] 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).

[20] 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.

[21] Beibin Li, Yao Lu, and Srikanth Kandula. 2022. Warper: Efficiently adapting learned cardinality estimators to data and workload drifts. In *Proceedings of the 2022 International Conference on Management of Data*. 1920–1933.

[22] 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.

[23] 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.

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

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

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

[27] 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 AAAI Conference on Artificial Intelligence*, Vol. 30.

[28] 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).

[29] 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.

[30] Zahra Sadri, Le Gruenwald, and Eleazar Leal. 2020. Online index selection using deep reinforcement learning for a cluster database. In *2020 IEEE 36th International Conference on Data Engineering Workshops (ICDEW)*. IEEE, 158–161.

[31] 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.

[32] 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.

[33] Ji Sun and Guoliang Li. 2019. An end-to-end learning-based cost estimator. *ArXiv Preprint ArXiv:1906.02560* (2019).

[34] 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.

[35] 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.

[36] Matias Valdenegro-Toro. 2019. Deep sub-ensembles for fast uncertainty estimation in image classification. *ArXiv Preprint ArXiv:1910.08168* (2019).

[37] Laurens Van der Maaten and Geoffrey Hinton. 2008. Visualizing data using t-SNE. *Journal of Machine Learning Research* 9, 11 (2008).

[38] Xiaoying Wang, Changbo Qu, Weiyuan Wu, Jiannan Wang, and Qingqing Zhou. 2020. Are we ready for learned cardinality estimation? *ArXiv Preprint ArXiv:2012.06743* (2020).

[39] Yeming Wen, Dustin Tran, and Jimmy Ba. 2020. Batchensemble: An alternative approach to efficient ensemble and lifelong learning. *ArXiv Preprint ArXiv:2002.06715* (2020).

[40] Lianggui Weng, Rong Zhu, Di Wu, Bolin Ding, Bolong Zheng, and Jingren Zhou. 2023. Eraser: Eliminating performance regression on learned query optimizer. *https://github.com/duoyw/Eraser/tree/main/paper* (2023).

[41] 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.

[42] 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.

[43] 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.

[44] Erkang Zhu, Dong Deng, Fatemeh Nargesian, and Renée J Miller. 2019. Josie: Overlap set similarity search for finding joinable tables in data lakes. In *Proceedings of the 2019 International Conference on Management of Data*. 847–864.

[45] 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.

[46] 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.

[47] 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.

[48] 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.