Low Rank Learning for Offline Query Optimization

Zixuan Yi
University of Pennsylvania
USA
zixy@cis.upenn.edu

Yao Tian
The Hong Kong University of Science and Technology
China
ytianbc@cse.ust.hk

Zachary G. Ives
University of Pennsylvania
USA
zives@cis.upenn.edu

Ryan Marcus
University of Pennsylvania
USA
rcmarcus@cis.upenn.edu

Abstract

Recent deployments of learned query optimizers use expensive neural networks and ad-hoc search policies. To address these issues, we introduce LimeQO, a framework for offline query optimization leveraging low-rank learning to efficiently explore alternative query plans with minimal resource usage. By modeling the workload as a partially observed, low-rank matrix, we predict unobserved query plan latencies using purely linear methods, significantly reducing computational overhead compared to neural networks. We formalize offline exploration as an active learning problem, and present simple heuristics that reduces a 3-hour workload to 1.5 hours after just 1.5 hours of exploration. Additionally, we propose a transductive Tree Convolutional Neural Network (TCNN) that, despite higher computational costs, achieves the same workload reduction with only 0.5 hours of exploration. Unlike previous approaches that place expensive neural networks directly in the query processing "hot" path, our approach offers a low-overhead solution and a no-regressions guarantee, all without making assumptions about the underlying DBMS.

CCS Concepts

• **Information systems → Query optimization**; **Query planning**; • **Mathematics of computing → Computations on matrices**.

ACM Reference Format:
Zixuan Yi, Yao Tian, Zachary G. Ives, and Ryan Marcus. 2025. Low Rank Learning for Offline Query Optimization. In *Proceedings of ACM SIGMOD Conference on Management of Data (SIGMOD '25).* ACM, New York, NY, USA, Article 183, 15 pages. https://doi.org/10.1145/3725412

$$\begin{array}{c|cccc|}
 & h_1 & h_2 & \dots & h_k \\
q_1 & 3 & 4 & \dots & ? \\
q_2 & 9 & ? & \dots & 6 \\
q_3 & ? & 15 & \dots & 3 \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
q_n & 5 & 1 & \dots & 2
\end{array}$$

Figure 1: An example *workload matrix.* Each row represents a query, and each column represents a hint. The value ? represents an unobserved latency.

1 Introduction

Recent advances in learned query optimization — using machine learning to completely replace or aid a traditional query optimizer [57] — have demonstrated significant performance gains [38, 73, 76, 79, 85]. However, learned optimizers also have several drawbacks: (1) the nature of learning techniques can cause *unpredictable regressions* (e.g., "my query was fast yesterday, why is it slow today?"), (2) they suffer from *expensive training and inference costs* [34] (e.g., from neural networks [39] or from training data collection times [73]), and (3) they often *make assumptions about the underlying DBMS*, such as the availability of features (e.g., cost estimates [38]) or the structure of query plans (e.g., tree structured plans with finite operators [76]).

In the context of repetitive analytic workloads, such as updating live dashboards and timely report generation, two recent works in production systems have addressed the first issue of unpredictable performance regressions: AutoSteer [3] and QO-Advisor [82]. The core idea behind both approaches is to use offline execution to verify that potential new query plans are actually better than the default plan. If verified, the new query plan is added to a plan cache and used when an eligible query arrives. This simple technique ensures that no query *ever* regresses (absent data shift), but at the cost of potentially expensive offline execution.

While major steps in the right direction, neither AutoSteer nor QO-Advisor are deeply strategic in their offline exploration: both exhaustively test a set of alternative plans, and control for excessive offline exploration time by heuristically limiting the set of alternative plans explored. For example, QO-Advisor limits the set of alternative plans to those produced by a "single rule flip" [82], while AutoSteer simply tests the *n* most promising plans. To the best of our knowledge, **the broader problem of how to effectively explore the space of alternative plans offline, in a way that maximizes workload benefit while minimizing offline computation time, has not been systematically explored.**

Authors' Contact Information: Zixuan Yi, University of Pennsylvania, USA, zixy@cis.upenn.edu; Yao Tian, The Hong Kong University of Science and Technology, China, ytianbc@cse.ust.hk; Zachary G. Ives, University of Pennsylvania, USA, zives@cis.upenn.edu; Ryan Marcus, University of Pennsylvania, USA, rcmarcus@cis.upenn.edu.

SIGMOD '25, Berlin, Germany

ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/3725412

Here, we formalize and expand on this offline exploration approach. Our proposed framework seeks to minimize offline resource usage while maximizing performance improvements, maintaining the "no-regressions" guarantee (compared to the underlying traditional optimizer) of prior work [3, 82]. Additionally, we avoid any tight coupling between our framework and specific DBMSes: we do not make assumptions about the structure of query plans, the number of operators, or even the availability of cost estimates. The only assumption our framework makes is that each query plan has a number of alternatives with measurable latency. To accomplish this, we introduce a new approach to learned query optimization called LimeQO, which is trained with *purely linear methods*, leading to drastically simpler and lower overhead implementations.

Target workload & hints. Like AutoSteer and QO-Advisor, we target query workloads that are mostly repetitive. Thus, we assume that *most* queries and their set of potential query plans are known ahead of time, although we do support the addition of new queries over time. Additionally, like prior work on learned query optimization [3, 38, 44, 73, 82], we assume that the underlying query optimizer provides a "hint" interface to create different variations of query plans. We justify these decisions in Section 3.

Workload matrix. Our core insight is to model the problem of offline optimization as a *matrix completion (MC)* [21] problem: we can represent a workload with n repeated queries and k possible query hints as a *workload matrix* $\mathbf{W}$, where each entry is the latency of a plan, as shown in Figure 1. Selecting the best hint for each query amounts to taking a row-wise minimum. Unfortunately, computing the whole matrix would require $n \times k$ query executions, which could be prohibitive. Instead, we consider the workload matrix to be *partially observed*: some entries are known (observed, or part of the training set), and other entries are unknown (unobserved). "Completing" the matrix means predicting the unobserved values.

Readers familiar with MC may note that one common application of MC is recommendation systems [51]. Indeed, we will show that LimeQO works for similar reasons as recommendation systems do: since sets of queries that perform well with some hints also tend to perform poorly with other hints, the *rank r* of the workload matrix is low. This **low rank** means that (among other things) we can construct an accurate estimate of $\mathbf{W}$ using two factored matrices $\mathbf{Q} \in \mathbb{R}^{n\times r}$ and $\mathbf{H} \in \mathbb{R}^{k\times r}$: $\hat{\mathbf{W}} = \mathbf{Q}\mathbf{H}^T$. This approximation can be found using purely linear methods that use *100x less computational resources than their neural network counterparts* (e.g., TCNNs [41]).

Offline optimization as active learning. Matrix completion allows us to approximate missing entries in the workload matrix, but we still need a way to explore the workload matrix efficiently. Ideally, we want to discover the minimum value of each row (the fastest hint for each query) as quickly as possible. This can be considered an *active learning* [53] problem, in which we must intelligently select which new pieces of information to observe next. Observing each new piece of information has an associated value (the amount we can improve the latency of the query) and a cost (the amount of offline exploration time we use). We present two simple algorithms inspired by active learning that are especially suited toward this special variant of the problem. These active learning techniques allow us to achieve near-optimal performance by spending only the default workload time (i.e., a few hours) rather than exhaustively exploring the entire space, which would take over 10 days.

Transductive neural networks. When computational overhead is not a limiting factor, and when certain assumptions (such as tree-structured plans) can be made about the underlying database system, our framework can also integrate expensive neural network models. Of course, doing so increases inference overhead, but may lead to faster convergence due to the power of neural networks. We present a new type of tree convolution neural network (TCNN) [41] called a *transductive TCNN* which combines the tree-structured inductive bias of a TCNN with *learned* representations of the $\mathbf{Q}$ and $\mathbf{H}$ matrices. Unsurprisingly, our computationally expensive neural network is a better approximator of $\mathbf{W}$ than purely linear methods. It accelerates the 3-hour workload by a factor of 2 with 0.5 hours of offline exploration, whereas purely linear methods took 1.5 hours of exploration to achieve the same speedup. However, the overhead of the neural network in the inference phase is 360 times higher than linear methods.

Trouble with timeouts. A key challenge to exploring the workload matrix $\mathbf{W}$ is dealing with queries with unusually long latencies. For a particular row in the matrix, if the current best query plan takes x seconds, then in some sense it is wasteful to execute any other plan in that row for longer than x seconds: once a plan takes longer than x seconds, we can rule it out as the optimal plan. Unfortunately, simply placing the timed out query value into $\mathbf{W}$ will mislead the machine learning model: it will look like the timed-out query plan took x seconds to execute, but in reality, that plan could have taken much longer (the true latency of the plan is not known, but the fact that true latency is greater than x is known). Prior work, like Balsa [76], has addressed this issue by setting the query timeout to some integer multiple S of x, allowing the model to at least see that the timed-out plan took longer than Sx to execute. However, this solution is still suboptimal, since (1) it executes the timed-out query for longer than necessary (i.e., by an integer factor) and (2) still "misleads" the machine learning model by treating the latency of the query as Sx.

In this work, we show how a well-studied machine learning trick called *censored observations* [71] can be applied to learned query optimization. Using our technique, we can treat timed-out queries as "first-class citizens," penalizing models for underestimating the the timed-out query's latency, but not penalizing the model for a (potentially valid) over-estimate. We show how to handle censored observations both in MC and in the transductive TCNN. Our experiment shows that the censored technique reduced the 3-hour workload to 1.5 hours after 0.5 hours of exploration, whereas without the censored technique, it took 0.9 hours to achieve the same reduction.

Contributions We make the following contributions:

- We present LimeQO, a framework for offline exploration for query optimization formalized as an active learning problem, and present two simple heuristic solutions.
- We present a modified version of the popular alternating-least-squares (ALS) [21] MC algorithm which can handle censored observations (timeouts).

- We present the transductive TCNN, a neural network specially designed for offline query optimization which takes advantage of the low rank structure of the workload matrix, which can also handle censored observations.
- We show how LimeQO can be extended to handle new queries and data drift.

The rest of this paper is organized as follows. We present our system model in Section 3. We define the offline exploration problem in Section 4. In Section 5, we present experimental results. Finally, we present related works in Section 2 and concluding remarks in Section 6.

2 Related Work

Learned Query Optimization. Recent works have explored the integration of machine learning techniques into several components in DBMS, such as learned cardinality estimators [18, 28, 30, 36, 46, 52, 77], learned cost models [23], and learned query optimizers. Learned query optimizers are broadly divided into two categories: "full" learned optimizers that synthesize entire query execution plans from scratch, effectively replacing the traditional query optimizer [11, 26, 32, 39, 40, 48, 76, 80, 85], and "steering" learned optimizers that sit on top of a traditional optimizer [12, 38, 73, 78, 79]. The latter "steering" approach has fewer degrees of freedom, but exhibits lower variation, leading to adoption in some production systems [3, 82, 86]. Since any performance variation can be harmful to downstream applications, offline execution is often used to verify performance improvements [3, 37, 68, 82], although confidence-learning based approaches are also being developed [25, 26, 72]. Our approach builds upon the "steering" approach. Unlike AutoSteer [3] and QO-Advisor [82], we explore the space of query-hints combinations *holistically*, taking the entire workload into account, reducing the need for exhaustive execution while still preventing regressions. To the best of our knowledge, the only prior work on learned query optimization at the workload level is GALO [15], which mines query logs for problematic executions and recommends fixes for slow queries. Notably, GALO does not require offline query execution like the current work, but GALO is not guaranteed to be regression-free.

Our approach is closely related to BayesQO [67], a concurrent method also aimed at offline query optimization. However, while BayesQO optimizes one query at a time, our framework simultaneously optimizes an entire query workload. Furthermore, while BayesQO considers the entire set of possible query execution plans, our method considers a small, finite set of execution plans (e.g., 49 candidate plans in PostgreSQL). By optimizing at the workload level instead of the individual query level, our approach can leverage inter-query similarities and the low-rank structure of the workload matrix. Additional details on the performance differences are provided in Section 5.6.

Matrix Completion. Matrix completion is a decades-old technique [19] that has been widely used in collaborative filtering [31] and recommendation systems [21], although MC has also been the subject of deep mathematical investigation [8]. Linear methods have also been used by learned cardinality estimators [63]. Several algorithms have been proposed for matrix completion, including nuclear norm minimization [9], singular value thresholding [7], and alternating least squares (ALS) [21]. More recently, deep learning techniques have been introduced to capture complex, non-linear relationships in recommendation systems [14, 20, 22, 43, 54, 81]. These methods leverage neural networks to learn intricate patterns in user-item interactions, outperforming traditional linear models. Our LimeQO+ approach differs from these existing models by using the transductive approach [47] that incorporate query plan trees into each matrix entry, rather than relying on the user and item features.

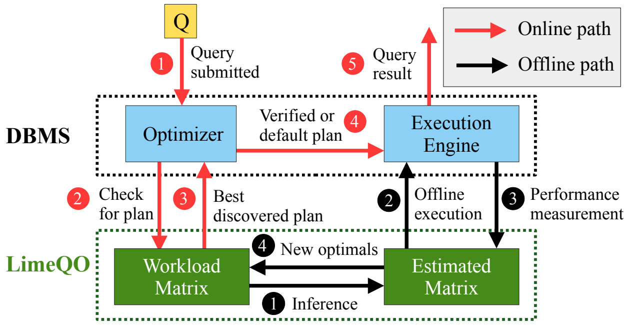

Figure 2: LimeQO system model

Active Learning. Active Learning [53, 59] is a well-established technique that aims to select the next training instance to label, with the primary goal of optimizing for the most information gain. As discussed in Section 4.2, most existing works assume that the cost of acquiring the label is fixed [10, 55, 65], irrespective of the instance, and thus does not need to balance between labeling cost and the value of the label. While Selective Supervision [27] does explore the idea of balancing the cost of labeling against expected improvement, it still assumes a uniform cost within each class, which is not valid in our case. Additionally, many multi-armed bandits-based active learning methods [6, 17, 24] assume an unlimited pool of observations, which also does not fit our scenario. We found these approaches ineffective for our matrix completion problem, highlighting the need for more customized strategies. Our Greedy strategy seeks to prioritize the longest-running query under the premise that it has the most potential value. In contrast, LimeQO uses its current predictive model to estimate the instance with the largest expected benefit.

3 System model

LimeQO uses offline exploration to find the optimal hint for a set of queries in a repetitive workload. LimeQO does this by formulating the problem as low-rank matrix completion. Leveraging the low-rank property allows us to efficiently and accurately complete the workload matrix **W**. LimeQO operates externally to the DBMS, interacting with both the query optimizer and the execution engine. It explores better plans within the available query hint space by "steering" [38, 73] the existing query optimizer. The system model for this interaction is shown in Figure 2.

Our prototype system was implemented to target the query optimizer for PostgreSQL, which features one of the most complex query optimizers in a widely used open source query engine. Numerous systems, including Amazon RedShift, TimescaleDB, and EnterpriseDB all build upon PostgreSQL's core, and our system would naturally extend to them. Additionally, the general approach of query hints is used by several other query optimizers, such as those in Presto [64], MySQL, and Microsoft's SQL Server and SCOPE [82].

Our framework has two paths: an online path, in which user-submitted queries are executed using plans that have been verified to be fast, and an offline path, where LimeQO can perform offline exploration. In the *online path*, ❶ user-submitted queries are received by the DBMS' traditional optimizer. Then, ❷ the optimizer asks LimeQO if a better query plan has been observed for this query. ❸ LimeQO replies with either a query plan that is faster than the default plan, or the default plan. ❹ This verified plan is then executed, ❺ and the results are returned. In the *offline path*, LimeQO searches for better query plans. This offline search could happen when the DBMS is idle [13], or could be performed on a snapshot of the database. During this time, LimeQO will ❶ predict the performance of all query plans in the workload matrix, and then ❷ select the most promising query plans to explore. Next, ❸ these promising plans are executed, using a timeout based on the current best-known plan for that query. Once the new query plan finishes executing or times out, ❹ the performance is recorded. ❺ The newly observed values are finally stored back into the workload matrix. Importantly, the workload matrix contains two types of entires: (a) *complete* entries, representing query plans whose latency was observed via execution, or *censored* entries, representing queries plans that timed out, but for which a lower bound on their execution time is now known (e.g., if a query plan times out after 2 minutes, we assume that the true latency of that plan is greater than 2 minutes).

Our goal. A naive implementation of LimeQO could simply evaluate random unobserved query plans (i.e., test blank entries in the workload matrix), but this strategy could waste offline execution time testing bad plans. Executing the entire matrix exhaustively is impractical; for example, processing the full CEB [45] workload would take 12 days, and the Stack [45] workload would require even longer than 16 days. Thus, LimeQO must strategically use each moment of offline execution time to create the largest improvement to the overall workload.

Assumptions. We make the following assumptions:

(1) The DBMS repeatedly executes a set of queries, referred to as the workload. Identical queries occur multiple times, and we cache them after the second occurrence and begin optimizing their performance. Thus, the workload consists of a set of unique queries.
(2) Each query in the workload has a default plan chosen by the underlying query optimizer, as well as a set of alternative plans, or hints [3, 38, 73], a commonly used technique to guide query optimization.
(3) Queries generally exhibit consistent performance, meaning that the measurements obtained during the offline phase closely mirror those during online execution. Some may argue that data shifts might occur between the offline and online stages, potentially affecting performance. To address this concern, we provide two key observations. First, we demonstrate that near-optimal performance can be achieved after a duration equivalent to the total workload time (Section 5.1), making the time gap between the two phases negligible. Second, we show that even if data shifts occur, the best hint typically remains the same (Section 5.4), thus not significantly impacting our performance gains.

Why target repetitive workloads? At first glance, considering only *repeating* queries may seem like a major restriction. While there are certainly workloads with few or no repeating queries, there are also workloads like live dashboards that are almost purely repetitive [3]. Recent studies of the AWS Redshift analytics database product found that more than 50% of the queries executed on the Redshift fleet were repeated within 24 hours [56, 75, 83], 75% are repeated within a week, and 80% within a month. Furthermore, analyses have shown that long-running queries (those taking longer than 1 hour) almost always repeat across all clusters [69]. In Microsoft's SCOPE database, over 60% of the job volume is recurring [82]. Thus, targeting repeated workloads is both practical and impactful.

Handling novel queries. However, focusing solely on repetitive queries is still a major limitation to practitioners: new queries might not come along very often, but new queries almost certainly are introduced over a long period of time. Thus, we can say that new queries are *rare but guaranteed.* LimeQO can support such rarely-arriving new queries in an intuitive way: we simply add new rows to the workload matrix. The prior entries in the workload matrix can potentially help predict the entries for the newly added rows (queries). The first time a new query is added, it is always executed using the underlying DBMS' default plan to avoid regressions, so one cell of the new rows is initialized. We evaluate and discuss LimeQO's performance on novel queries in Section 5.3.

Why use query steering / hints? We assume, like prior work [3, 38, 44, 73, 82], that the underlying DBMS' query optimizer supports "hints" that change the behavior of the optimizer. Each hint is a coarse-grained knob in the optimizer that impacts query selection, for example, disabling or enabling a particular join operator. For simplicity, we use "hint" to refer to a specific configuration of the optimizer, which in some systems may mean a combination of multiple distinct hints (i.e., "hint sets" [38]).

Casting learned query optimization as a hint selection problem, first done by Bao [38], as opposed to fully replacing a query optimizer with a learned component [39, 76, 85], or deeply integrating a solution with a particular optimizer, has a number of advantages. First, the "hinting" interface as described is implemented by a wide variety of databases, including PostgreSQL [50], Presto [64], SCOPE [44], and RedShift [4]. Second, while optimizer hints are coarse-grained, they are also robust: selecting a plan that has been generated by a traditional query optimizer with a particular hint is much more likely to result in a reasonable plan than finer-grained techniques [38]. Third, hints have *enough* granularity to significantly improve a wide variety of analytic queries [44], making hint steering an especially good match for repetitive analytic workloads.

4 Offline Exploration

In this section, we first formulate the offline exploration problem (Section 4.1). Then we introduce our active learning exploration policy (Section 4.2), and extend it with two predictive models: a linear method (Section 4.3.1) and a neural method (Section 4.3.2).

4.1 Problem Definition

Formulation. Let $Q = \{q_1, \ldots, q_n\}$ be a set of regularly executed queries, and let $H = \{h_1, \ldots, h_k\}$ be a set of hints. We define a workload matrix $\mathbf{W}$ as a $n \times k$ matrix that holds the performance metric (e.g., latency) for each query (row) and for each hint (column): that is, $\mathbf{W}_{ij}$ represents the latency of running query q_i with hint h_j. Since exactly computing $\mathbf{W}$ is prohibitive (i.e., requiring nk query executions), we assume we only have access to a *partially observed* copy of $\mathbf{W}$, denoted as $\tilde{\mathbf{W}}$:

$$\tilde{\mathbf{W}}_{ij} = \begin{cases} \mathbf{W}_{ij} & \text{if } \mathbf{W}_{ij} \text{ is observed} \\ \infty & \text{otherwise} \end{cases} \tag{1}$$

When a query $q_i \in Q$ arrives, we select the hint h_j with the best observed latency, that is, the minimum value in the row $\tilde{\mathbf{W}}_i$.

Our goal is to design an *exploration policy* to reveal unobserved entries that can optimize performance while minimizing the offline time spent revealing entries of $\tilde{\mathbf{W}}$. We define P as the current workload latency,[1] (i.e., the sum of the minimum observed values for each query) as follows:

$$P(\tilde{\mathbf{W}}) = \sum_{i=1}^{n} \min_{1 \le j \le k} \tilde{\mathbf{W}}_{ij} \tag{2}$$

and we define T as the offline exploration time required for revealing entries in the matrix to attain $\tilde{\mathbf{W}}$:

$$T(\tilde{\mathbf{W}}) = \sum_{i=1}^{n} \sum_{j=1}^{k} \tilde{\mathbf{W}}_{ij} \cdot \mathbf{1}_{\{\tilde{\mathbf{W}}_{ij} \neq \infty\}} \tag{3}$$

The challenge lies in minimizing both the workload latency $P(\tilde{\mathbf{W}})$ and the total offline exploration time $T(\tilde{\mathbf{W}})$ simultaneously. While $P(\tilde{\mathbf{W}})$ can be minimized by fully exploring all entries of $\mathbf{W}$, this would maximize $T(\tilde{\mathbf{W}})$. Conversely, $T(\tilde{\mathbf{W}})$ can be trivially independently minimized by doing no exploration at all, leading to sub-optimal $P(\tilde{\mathbf{W}})$ performance. Thus, we seek an algorithm to achieve both objectives simultaneously.

Timeouts. A key aspect of the offline exploration process is that we are only interested in hints that outperform the current best observed hint. Therefore, we can safely optimize exploration by introducing a timeout limit, $\mathbf{T}_{ij}$, for each entry in the matrix, which is set to the current minimum latency observed in the corresponding row of $\tilde{\mathbf{W}}$:

$$\mathbf{T}_{ij} = \min_{1 \le j \le k} \tilde{\mathbf{W}}_{ij} \tag{4}$$

Algorithm 1: LimeQO

```
Input: $\tilde{\mathbf{W}}$: initial observed matrix; $\mathbf{M}$: mask matrix; $\mathbf{T}$:
       timeout matrix; $pred$: predictive model
Output: Hint selections $[h_1, \ldots, h_n]$ for workload
while $\mathbf{M} \neq \mathbf{1}$ do
    $\hat{\mathbf{W}} \leftarrow pred(\tilde{\mathbf{W}}, \mathbf{M}, \mathbf{T})$;
    for $i = 1$ to $n$ do
        $h_j \leftarrow H[\text{argmin}_j(\hat{\mathbf{W}}_{ij})]$;
        $ri \leftarrow (\min \tilde{\mathbf{W}}_i - \hat{\mathbf{W}}_{ij})/\hat{\mathbf{W}}_{ij}$;
        add $(q_i, h_j)$ to $S$ if $r_i > 0$ ;
    Select top $m$ largest $(q_i, h_j)$ from $S$ w.r.t. $r_i$;
    if not enough to select then
        randomly select some unobserved $(q_i, h_j)$;
    $\mathbf{T}_{ij} = \min(\min(\tilde{\mathbf{W}}_i), \hat{\mathbf{W}}_{ij} \times \alpha)$;
    Offline execute, timeout if $\mathbf{W}_{ij} \geq \mathbf{T}_{ij}$;
    Update $\mathbf{M}$, $\mathbf{T}$, and $\tilde{\mathbf{W}}$;
for $i = 1$ to $n$ do
    $h_i \leftarrow H[\text{argmin}_j(\tilde{\mathbf{W}}_{ij})]$;
return $[h_1, \ldots, h_n]$
```

This allows us to update the workload matrix $\tilde{\mathbf{W}}$ by applying the timeout condition:

$$\tilde{\mathbf{W}}_{ij} = \begin{cases} \mathbf{W}_{ij} & \text{if } \mathbf{W}_{ij} \text{ is observed} \\ \mathbf{T}_{ij} & \text{if } \mathbf{W}_{ij} \text{ exceeds the timeout} \\ \infty & \text{otherwise} \end{cases} \tag{5}$$

We note that $\tilde{\mathbf{W}}$ evolves dynamically during the execution of the algorithm, as new entries in the matrix are observed.

By incorporating timeouts, we can bound the time spent on exploration of hints that will show no performance improvement – even if our predicted exploration time was incorrect.

4.2 Active Learning on a Low-Rank Matrix

Active learning strategies can be employed to efficiently explore and reveal unobserved entries in the matrix. Instead of exhaustively probing all entries, an active learning approach aims to identify the most informative entries to query, which can significantly reduce the exploration cost.

Here we propose two active learning techniques: Greedy and LimeQO (Algorithm 1).

Greedy. Greedy does not rely on any predictive model. It selects the queries with the largest current minimum observed latency, ie. $\text{argmax}_i(\min_{1 \le j \le k} \tilde{\mathbf{W}}_{ij})$. Then for each query, we randomly select an unobserved hint. This strategy focuses on improving queries with the worst observed performance, as they offer the greatest potential for reducing the overall workload latency $P(\tilde{\mathbf{W}})$.

The underlying assumption of the greedy technique is that there is a correlation between the *duration of a query plan* and that query plan's potential *room for improvement.* While this assumption is often true in academic benchmarks (where we select long-running queries precisely because they have a lot of room for improvement

[1] Practitioners may also be interested in optimizing tail latency instead of total latency, in which case P can be defined as the tail latency of the workload.

— otherwise the benchmark would not be very interesting), this assumption might not be true in practice. For example, the longest-running queries on Amazon RedShift are normally `COPY` queries or ETL jobs [70] (e.g., a query that dumps the result of a simple scan to a CSV file). These types of queries have almost no room for improvement, since they are almost entirely bounded by write speed. We demonstrate this experimentally in Section 5.1. Nevertheless, we evaluate Greedy as a useful baseline.

LimeQO. The LimeQO approach, on the other hand, uses a predictive model to guide exploration. We will first use the predictive model to complete the partially observed matrix $\tilde{\mathbf{W}}$ and generate the predicted matrix $\hat{\mathbf{W}}$, which fills in the unobserved entries using estimated values. Then, LimeQO selects query plans with the largest expected benefit, balancing the minimization of both $P(\tilde{\mathbf{W}})$ and the offline exploration time $T(\tilde{\mathbf{W}})$. For each query q_i, we compute the expected improvement ratio as follows:

$$r_i = \left(\min_{1 \le j \le k} \tilde{\mathbf{W}}_{ij} - \min_{1 \le j \le k} \hat{\mathbf{W}}_{ij} \right) / \min_{1 \le j \le k} \hat{\mathbf{W}}_{ij} \tag{6}$$

This ratio captures the potential performance improvement from exploring the predicted best hint for query q_i compared to its current best observed hint. By normalizing with the predicted best latency, we ensure that exploration focuses on minimizing both $P(\tilde{\mathbf{W}})$ and $T(\tilde{\mathbf{W}})$.

Algorithm 1 presents LimeQO in detail: Given an initial $\tilde{\mathbf{W}}$, we use a predictive model to construct an estimate $\hat{\mathbf{W}}$ (Line 2). With the estimated value, we go through every row of the predicted matrix and compute *expected improvement ratio* (Equation 6 and Line 6). The top m queries (Line 7), based on this improvement ratio, are then selected for exploration. In the case where there are fewer than m positive predicted improvements (Line 8), we will randomly select some unobserved entries (Line 9) to observe. Finally, we execute the m selected plans, timeout the plan if the plan's latency is greater than $\mathbf{T}_{ij}$ (Line 11), record their latency, and update $\mathbf{M}$, $\mathbf{T}$, and $\tilde{\mathbf{W}}$ (Line 12). This process can be repeated until there is no more offline exploration time left, or when the algorithm stops finding potential improvements. Finally, we will return the current best hint for each query (Line 13-14).

Why not use existing active learning approaches? Our two proposed methods join an extensive literature of methods within the active learning community [53, 59]. Surprisingly, many existing active learning techniques make fundamental assumptions that render them unsuitable for our environment. For example, most of the techniques specific to active matrix completion assume that the cost of acquiring the label is fixed [10, 55, 65], and thus there is no need to balance the potential improvement in query time versus the cost of exploring a query hint. Additionally, the scoring function in the Bayesian active learning method [60] are less suitable for the unique characteristics of our problem space, since in our problem the cost of revealing a matrix cell is correlated with the matrix cell's value (i.e., the cost of revealing the plan is the plan latency). Thus, we intentionally designed our scoring function to prioritize the efficient discovery of optimal hints for query optimization. Even methods such as selective supervision [27], which balance cost against improvement, assume a uniform cost within each class. The few methods we found that do not assume a fixed value or fixed cost were based on multi-armed bandits [6, 17, 24, 66], which assume that the pool of unlabeled observations is so large that taking an unlabeled observation and labeling it does not change the distribution of unlabeled observations. This is patently untrue in our scenario, since observing a matrix entry obviously prevents that same matrix entry from being observed again. We tested each of these approaches to see if they would work even if their core assumptions were violated, but we were not successful.

Algorithm 2: ALS

```
Input: $\tilde{\mathbf{W}}$: observed matrix; $\mathbf{M}$: mask matrix; $\mathbf{T}$: timeout
       matrix; $r$: rank; $\lambda$: regularization parameter; $t$:
       number of iterations
Output: Completed matrix $\hat{\mathbf{W}}$
Initialize $\mathrm{Q}, \mathrm{H}$ of size $\mathrm{n} \times \mathrm{r}$, and $\mathrm{k} \times \mathrm{r}$ randomly ;
for $i = 1$ to $t$ do
    $\hat{\mathrm{W}} \leftarrow \mathrm{M} \odot \tilde{\mathrm{W}} + (1 - \mathrm{M}) \odot \mathrm{QH}^T$
    if $\hat{\mathrm{W}} < \mathrm{T}$ and $\mathrm{T} > 0$ then
        $\hat{\mathrm{W}} = \mathrm{T}$ // Censored technique
    $\mathrm{Q} \leftarrow \hat{\mathrm{W}}\mathrm{H}(\mathrm{H}^T\mathrm{H} + \lambda I)^{-1}$ // Update Q with least squares solution
    $\mathrm{Q}[\mathrm{Q} < 0] = 0$ // Ensure non-negative entries
    $\hat{\mathrm{W}} \leftarrow \mathrm{M} \odot \tilde{\mathrm{W}} + (1 - \mathrm{M}) \odot \mathrm{QH}^T$
    if $\hat{\mathrm{W}} < \mathrm{T}$ and $\mathrm{T} > 0$ then
        $\hat{\mathrm{W}} = \mathrm{T}$ // Censored technique
    $\mathrm{H} \leftarrow \hat{\mathrm{W}}\mathrm{Q}(\mathrm{Q}^T\mathrm{Q} + \lambda I)^{-1}$ // Update H with least squares solution
    $\mathrm{H}[\mathrm{H} < 0] = 0$ // Ensure non-negative entries
$\hat{\mathrm{W}} \leftarrow \mathrm{M} \odot \tilde{\mathrm{W}} + (1 - \mathrm{M}) \odot \mathrm{QH}^T$
return $\hat{\mathrm{W}}$
```

4.3 Predictive Model

In real-world scenarios, the workload matrix $\mathbf{W}$ often exhibits low-rank structure due to inherent correlations among queries and hints. Leveraging this property, we propose two predictive models: linear methods based on matrix completion [21] (Section 4.3.1) and neural methods based on low rank embeddings and tree convolution neural networks (TCNN) [39, 41] (Section 4.3.2). These two models can be easily integrated as the predictive model in Algorithm 1.

4.3.1 Linear Method. We apply *matrix completion* [21] as the linear method. By assuming that $\mathbf{W}$ has low rank, the observed entries of $\mathbf{W}$ can be used to predict the unobserved entries. Notably, this technique uses the partially observed matrix $\tilde{\mathbf{W}}$ directly, and does not rely on any properties of the queries or their plans (e.g., cost estimates, plan structure, operators).

Matrix completion. Matrix completion (MC) is a technique used to recover unobserved entries in a low rank matrix [8, 9, 21, 61]. We define $\mathbf{M}$ as the *mask matrix*, which has the same shape as $\tilde{\mathbf{W}}$: $\mathbf{M}_{ij} = 0$ if $\tilde{\mathbf{W}}_{ij} = \infty$ and $\mathbf{M}_{ij} = 1$ otherwise (that is, $\mathbf{M}$ is one for observed entries of $\tilde{\mathbf{W}}$ and zero otherwise). Given a partially observed $\tilde{\mathbf{W}}$, a rank constraint r, and a regularization parameter λ, we can build

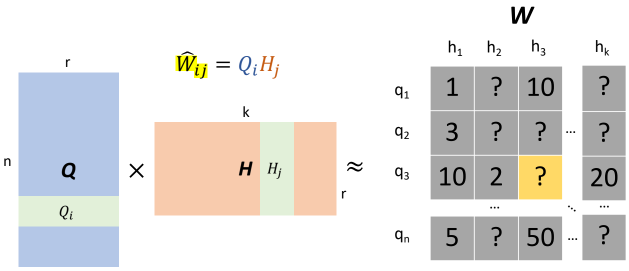

Figure 3: Linear Method

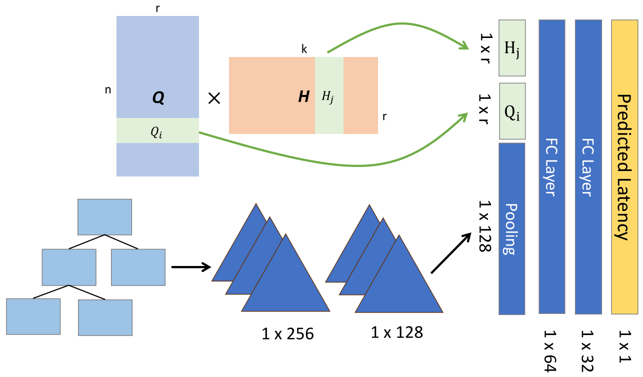

Figure 4: Neural Method

an estimate of $\mathbf{W}$ as $\hat{\mathbf{W}} = \mathbf{Q}\mathbf{H}^T$ by solving:

$$\min_{\mathbf{Q},\mathbf{H}} \left[\|\mathbf{M} \odot (\tilde{\mathbf{W}} - \mathbf{Q}\mathbf{H}^T)\|_F^2 + \lambda \left(\|\mathbf{Q}\|_F^2 + \|\mathbf{H}\|_F^2 \right) \right] \tag{7}$$

where $\mathbf{Q}$ and $\mathbf{H}$ are $n \times r$ and $k \times r$ matrices, respectively, and $\odot$ represents the element-wise product. To find $\mathbf{Q}$ and $\mathbf{H}$, we use the Alternating Least Squares (ALS) algorithm [21], which is based on the following key observation: while Equation 7 is not jointly convex in $\mathbf{Q}$ and $\mathbf{H}$, Equation 7 is convex in $\mathbf{Q}$ for fixed $\mathbf{H}$, and vice versa. Convexity in this context implies that optimizing $\mathbf{Q}$ or $\mathbf{H}$ while the other is fixed ensures convergence to a global minimum for that subproblem. This property is crucial to the ALS algorithm, which alternatives between solving each convex subproblem, progressively improving the overall solution (although there is no guarantee that the final approximation is optimal).

Algorithm 2 details our modified version of the ALS algorithm: we iteratively update the matrices $\mathbf{Q}$ and $\mathbf{H}$ using the least squares solution (lines 6 and 11), and fill in the predicted entries (lines 3 and 8). Additionally, since query latencies are strictly positive, we impose non-negative constraints on $\mathbf{Q}$ and $\mathbf{H}$ after each iteration (lines 7 and 12). While this constraint may slightly reduce approximation flexibility, it ensures the data remains physically meaningful (i.e., positive), allowing the score function (Equation 6) to operate effectively. The non-negative constraint can be interpreted as a heavy-handed[2] prior that query latency must be positive.

Predicted Latency. We illustrate the matrix factorization process in Figure 3. To calculate an unobserved entry, we simply compute the dot product of the corresponding two vectors: $\hat{\mathbf{W}}_{ij} = \mathbf{Q}_i\mathbf{H}_j$. Thus, each row of $\mathbf{Q}$ represents a "query vector" that contains information about the query in a particular row, and each column of $\mathbf{H}$ represents a "hint vector" that contains information about the hint in a particular column. We pick $\mathbf{Q}$ and $\mathbf{H}$ such that the dot product of a query vector and a hint vector predicts the latency of a given query under that specific hint.

Timeouts / Censored Technique. To handle the timeouts during the exploration process, we incorporate a censored technique specifically designed for this scenario. We define the timeout matrix $\mathbf{T}$, with the same shape as $\mathbf{W}$, where only the timeout values are filled in, and all other entries are zeros. As shown in Algorithm 2, lines 5 and 10, if the predicted value for a timed-out entry does not meet the timeout threshold (i.e., it is smaller than the timeout), we manually set it to the timeout value to reflect the observed limitations. This way, future iterations of ALS will never try to predict a value less than the timeout, but if a prior iteration of ALS predicts a value greater than the timeout, that value will be kept for future iterations.

4.3.2 *Neural Method.*

Here, we present an alternative approach for offline query optimization using a neural network. In this approach, we assume query plan features are available (e.g., cost and cardinality estimates), and that the underlying query optimizer generates tree-structured plans, such as those generated by PostgreSQL [1]. Leveraging these additional features may provide better accuracy than LimeQO, albeit with increased computational overhead.

Systems like Neo [39], Bao [38], and Balsa [76] use plan features to learn a value function that estimates the overall cost or latency of executing a query. Similarly, we propose a new predictive *transductive TCNN model*, which combines tree-structured features with the low-rank property of the workload matrix. We will first describe Tree Convolution [41], followed by an explanation of how TCNN Embedding integrates Tree Convolution with low-dimensional embeddings.

Tree Convolution. Tree convolution [41] is an adaption of traditional image convolution, designed for tree-structured data. It applies tree-shaped filters to query plans, identifying patterns related to query performance [39].

We first binarize the query plan trees as described in Bao [38], and encode each tree node into a vector that includes: 1) a one-hot encoding of the operator, 2) cost and cardinality information. After the final layer of tree convolution, dynamic pooling and fully connected layers are used to predict query performance. Tree Convolutions can effectively capture structural patterns in query plans and serves as an inductive bias for solving query optimization problems (i.e., the structure of the TCNN network is biased towards learning features that are useful for query optimization [39]). This technique has been widely applied in query optimization [3, 26, 38, 76].

TCNN Embedding. In contrast to the linear model approach (Section 4.3.1), where two matrices are used to replace a neural

[2] We say "heavy-handed" because negative entries in $\mathbf{Q}$ or $\mathbf{H}$ do not necessarily result in negative predicted latencies.

model and predict performance, TCNN Embedding combines low-dimension matrix representations with plan tree features using a neural network architecture. The key component of this approach is the **Embedding** layer, which provides compact vector representation of queries and hints.

As shown in Figure 4, each query and each hint is mapped to a vector of size r via the two embedding layers. These vectors are learned in such a way that similar queries and hints are represented by similar vectors. After retrieving the embedding vectors, we concatenate them with the outputs from the tree convolution layer. The concatenated vector, containing both structural information from the query plan tree and low-rank embeddings, is then passed through fully connected layers for performance prediction. This hybrid approach combines the advantages of tree convolution and low-rank embeddings, resulting in better performance compared to using either method alone.

The reason that the transductive TCNN captures the low-rank structure of the workload matrix is that the learned embeddings, labeled $\mathbf{Q}$ and $\mathbf{H}$ in Figure 4, are isomorphic to the linear decomposition matricies $\mathbf{Q}$ and $\mathbf{H}$: the embedding for a particular query represents features for the entire row of the matrix, and the embedding for a particular hint represents features for an entire column of the matrix. Since the same query embedding is used for every entry in a row of $\mathbf{W}$, and since the same hint embedding is used for every entry in a column of $\mathbf{W}$, the transductive TCNN can be said to have *weight sharing* [33].

Predicted Latency. In each offline exploration step, we train a TCNN Embedding model to predict the unobserved values in $\tilde{\mathbf{W}}$. Specifically, the model is trained using the observed entries of $\tilde{\mathbf{W}}$, using features extracted from each query plan as well as the corresponding query and hint indices (i, j). After training, the model performs inference to generate predicted latencies for the unobserved entries. Consequently, $\hat{\mathbf{W}}$ consists of the actual latencies for the observed entries and the model's predictions for the unobserved ones. Furthermore, the model is initialized with the weights from the previous step, enabling it to build on prior learning.

Timeouts / Censored Technique. Recall that for some entries in the matrix, we timed out at a certain threshold τ and thus the values represent a lower bound of the actual execution time. To effectively incorporate these censored observations into our neural network model, we introduce a new loss function specifically for the timed-out values in the neural network model training process:

$$\mathcal{L}(\hat{y}, y, \tau) = \frac{1}{n} \sum_{i=1}^{n} \mathbf{1}_{\{\hat{y}_i < \tau_i\}} \cdot (\hat{y}_i - y_i)^2 \tag{8}$$

where $\hat{y}_i$ is the predicted value, y_i is the true value, and τ_i is the timeout value for the ith entry. The term $\mathbf{1}_{\{\hat{y}_i < \tau_i\}}$ is an indicator function, which is 1 if predicted value is less than the threshold and 0 otherwise. This loss function ensures that only predictions less than the timeout threshold contribute to the loss calculation, penalizing the model for incorrect predictions where it is certain to miss the true value, while not penalizing it for predictions where the correctness of prediction is uncertain.

By replacing the standard Mean Squared Error (MSE) loss with this censored loss function, we enable the model to appropriately handle timeout observations and effectively learn from them.

5 Experiments

In this section, we conduct a series of experiments to evaluate our proposed techniques: LimeQO, the linear method introduced in Section 4.3.1, and LimeQO+, the neural method introduced in Section 4.3.2. Our experimental results seek to answer the following questions:

- How does LimeQO and LimeQO+'s performance compare to simple baselines and existing techniques? (Section 5.1) And how much overhead do LimeQO and LimeQO+ entail? (Section 5.2)
- How well does LimeQO handle workload shift (Section 5.3) and data shift? (Section 5.4)
- How reasonable is our low rank assumption, and how much does each component and optimization of LimeQO matter (ablation studies)? (Section 5.5)

Experimental setup. We evaluated LimeQO using PostgreSQL 16.1 [1]. LimeQO uses the same 49 hints as Bao [38], which are based on six configuration parameters where we can enable or disable hash join, merge join, nested loop join, index scan, sequential scan, and index-only scan[3]. Each (query, hint) pair is executed five times on an AMD Ryzen 5 3600 6-Core Processor, running Arch Linux 6.11.3. For the subsequent experiments, we selected the median runtime for each pair. Additionally, each technique's experiments were repeated five times, and we report the average runtime along with the standard deviation.

Workload and datasets. We evaluated the performance of LimeQO using multiple diverse workloads from prior work, which are detailed in Table 1. The **JOB** workload [35] is relatively small, comprising only 113 queries on the IMDb database. The **CEB** workload [45] expands upon JOB by adding thousands of additional queries. The **Stack** dataset contains over 18 million questions and answers from StackExchange websites (e.g., StackOverflow.com) collected over ten years. We have two snapshots of this data from 2017 and 2019. In Section 5.1, we use the 2019 version, and in Section 5.4, we utilize both versions to model data shifts. The **DSB** benchmark [16] is adapted from the TPC-DS benchmark [42] with more complex data distribution and more varieties in queries. Specifically, we select a scale factor of 50 (following prior work [74, 84]) and generate 20 distinct parameterized query instances from each query template.

For each workload, we list the time it takes for a vanilla PostgreSQL database to execute every query ("Default"), and the theoretical minimum time achievable by a not-possible-in-practice oracle function ("Optimal"). Each workload has between 1.36x to 2.66x "headroom" (Default/Optimal).

Techniques and tests. We compare six different methods. For each method, we initially reveal the entries in the workload matrix corresponding to the default plan produced by PostgreSQL, simulating an environment where queries are executed repeatedly.

[3] It is not possible to turn off all join operators or turn off all scan operators, hence 49 hints instead of 64.

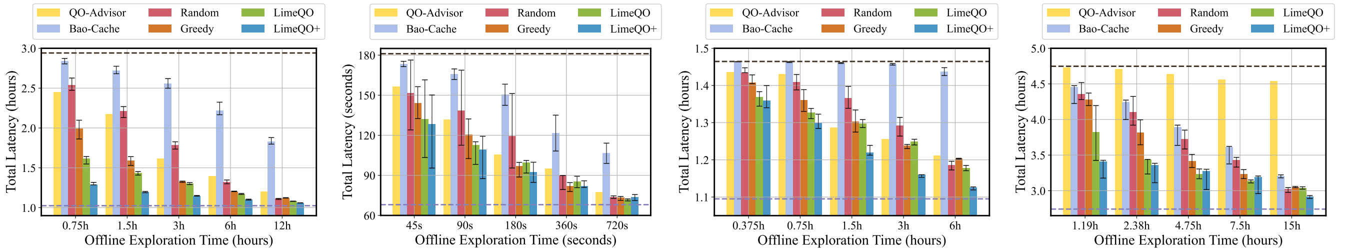

Figure 5: Performance Improvements measured on four different workloads: CEB, JOB, Stack, and DSB. Goal is to achieve lower total latency with less offline exploration time. Offline exploration times chosen on the X-axis correspond to [1/4, 1/2, 1, 2, 4] × default workload time in each workload.

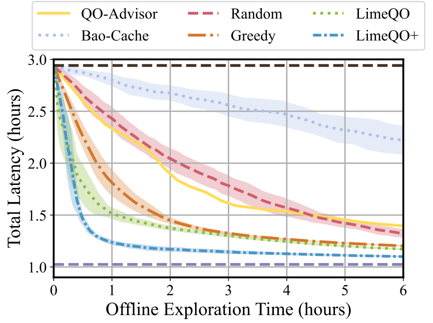

Figure 6: Total Latency vs. Offline Exploration Time on CEB workload. LimeQO and LimeQO+ outperform other techniques.

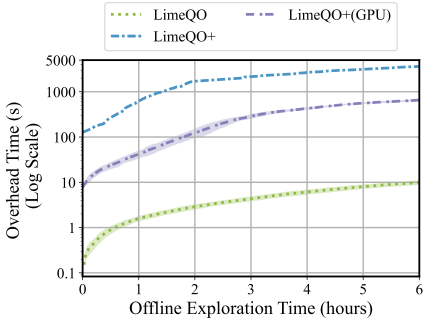

Figure 7: Overhead Time vs. Offline Exploration Time on CEB workload. On CPU, LimeQO+ spend 360x overhead time than LimeQO.

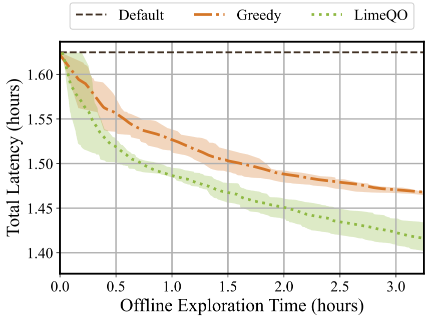

Figure 8: Greedy vs. LimeQO after we add a ETL query into the Stack Workload. Note that the default workload time increased from 1.46 hours to 1.62 hours.

Workload	Dataset	Size	# Queries	Default	Optimal
JOB [35]	IMDb	7.2 GB	113	181 s	68 s
CEB [45]	IMDb	7.2 GB	3133	2.94 hrs	1.02 hrs
Stack [38]	Stack	100 GB	6191	1.46 hrs	1.09 hrs
DSB [16]	DSB	50 GB	1040	4.75 hrs	2.74 hrs

Table 1: Four workloads we covered in the experiments. Default refers to the total time taken with PostgreSQL's default hint, while Optimal is the best time achievable if all hints were explored.

- QO-Advisor: the QO-Advisor technique [82] adapted to PostgreSQL. Instead of using a contextual bandit model that learn from the estimated cost to recommend single-rule flips, we select the unexplored entry with the lowest optimizer cost (this is the best action that QO-Advisor's contextual bandit could possibly pick, since QO-Advisor's multi-armed bandit operated over the optimizer's cost model).
- Bao-Cache: the technique of Bao [38] adapted to offline exploration. The TCNN is used to select unobserved entries to explore. We cache the results and select the best observed hint for each query (thus guaranteeing that there are no query regressions).
- Random: explore the workload matrix by randomly selecting unobserved entries.
- Greedy: explore the matrix by selecting the longest running queries as described in Section 4.2, then randomly picking the unobserved hints.
- LimeQO: use MC to explore the matrix as described in Section 4.3.1. We set $r = 5$, $\lambda = 0.2$ and $t = 50$ in Algorithm 2. We implemented it using standard linear algebra libraries, specifically NumPy's numpy.linalg which uses LAPACK [2] at core.
- LimeQO+: use TCNN Embedding as described in Section 4.3.2. For the TCNN component, we use the same TCNN architecture as [38], except that we add a dropout layer [62] with $p = 0.3$ between each tree convolution layer, which universally improved results. For the embedding layer, we set $r = 5$. Training is performed with Adam [29] using a batch size of 32, and is run for 100 epochs or convergence (defined as a decrease in training loss of less than 1% over 10 epochs) is reached.

5.1 Performance Improvements

How much can LimeQO and LimeQO+ improve latency? Figure 5 shows the total workload time across different workloads after a certain amount of offline exploration time.

On the **CEB** workload, after 1.5 hours (50% of the default workload time), LimeQO brings the latency down by 50%, from 2.94 hours to 1.45 hours — within 15% of the optimal reduction (65%). LimeQO+ achieves a 60% reduction, lowering the time from 2.94 hours to 1.2 hours, just 5% above the optimal. The relatively poor performance of Random and Greedy indicates that these improvements are not due to chance. For the **JOB** workload, after one default workload time, LimeQO and LimeQO+ reduce processing time to 100s (a 45% reduction) and 80s (a 56% reduction) respectively, whereas the optimal is 68s (a 62% reduction). On the **Stack** workload, after one default workload time, LimeQO and LimeQO+ reduce latency by 11% and 17% respectively, which is close to the optimal possible improvement of 25%. For the **CEB** workload, after one default workload time, LimeQO and LimeQO+ reduce latency to 3.25 hours (a 32% reduction), and 3.26 hours (a 31% reduction) respectively, whereas the optimal latency is 2.74 hours (a 42% reduction).

We observe that, across different workloads, LimeQO+ can achieve better results than LimeQO in many cases, but this comes at the cost of additional overhead (further investigated in Section 5.2). Both LimeQO and LimeQO+ outperforms Random and Greedy techniques at the start (0 to 1 × default workload time), although the four techniques converge after the 4 × default workload time. Compared to QO-Advisor and Bao-Cache, our techniques consistently demonstrated better performance across various exploration durations and workload settings, highlighting the importance of considering the entire workload at once when making exploration decisions.

Performance over time. We further analyze the CEB workload in Figure 6, where we show how different techniques' workload time changes with time spent on offline optimization.

Initially, LimeQO reduces workload latency more rapidly than LimeQO+. However, after approximately 20 minutes of exploration, LimeQO+ surpasses LimeQO, achieving a lower total latency. This shift can be attributed to LimeQO+'s deep learning approach, which improves its performance as it receives more training data.

Does Greedy always work well? The reader may notice that, in Figure 5's Stack workload results, Greedy and LimeQO both achieve around 1.3 hours at 1× the default workload time and approximately 1.24 hours at 2× the default workload time. While these results suggest that Greedy and LimeQO have comparable performance in this instance, this does not imply that the Greedy approach is universally effective. The Greedy method operates under the assumption that there is a correlation in the workload, such that longer-running queries have greater potential for performance improvement. However, this assumption may not hold true in real-world workloads. In fact, the performance of Greedy can significantly degrade in certain scenarios. To illustrate this, we conduct an experiment where we add a simple ETL query to the Stack workload. This ETL query loads the joined results of the `question` and `user` tables from the Stack database into a CSV file, which takes 576.5 seconds to execute. It is obvious that changing query optimizer hints will not reduce the runtime of this ETL query.

Figure 8 shows that from 0 to 3.25 hours (2x default workload time), LimeQO is consistently better than Greedy. This is because while Greedy persistently explore the long ETL query at each exploration step – because it is one of the longest-running queries in the workload, LimeQO utilizes the predictive model to recognize that the potential gain from optimizing this query is low. As a result, LimeQO intelligently ignores the ETL query and explore other queries where performance improvements are more attainable. This highlights the advantage of incorporating predictive modeling into the exploration strategy.

5.2 Overhead

Figure 7 shows the cumulative overhead time cost for LimeQO and LimeQO+ during offline exploration time on the CEB workload. In this context, the offline exploration time is the time DBMS spends on executing the queries. The overhead for LimeQO is the computational cost of matrix completion, while for LimeQO+, it includes both training the model on observed plan trees and inference on unobserved plan trees at each exploration step.

After exploring for 6 hours, LimeQO incurs a total overhead of just 10 seconds, whereas LimeQO+ experiences an overhead of approximately 3600 seconds (60 minutes). This indicates that linear methods are at least **360x** more efficient in terms of computational resource usage.

We also experiment with LimeQO+ on an NVIDIA A100 GPU, tuning training and inferencing batch size to 512 for optimal performance. Even with this powerful GPU, LimeQO+ still requires 660 seconds (11 minutes) that add to overall query processing overhead.

In conclusion, LimeQO+ requires significantly more resources than LimeQO. Additionally, it's important to note that the implementation of LimeQO+ is more complex and has a large software footprint (e.g., requiring PyTorch [49]) as well as feature extraction steps. On the other hand, LimeQO's implementation only requires near-universal linear algebra routines and straightforward filling of the observed latency time.

5.3 Workload Shift

Practitioners may also concerned LimeQO's ability to handle new queries, particularly in the event of a workload shift. A key question is whether LimeQO remains robust under these conditions.

In Figure 9, we evaluate the performance of LimeQO and Greedy under a workload shift using the CEB workload. From 0 to 2 hours, we process 70% of the full set of queries (randomly chosen). At the 2-hour mark, we introduce the remaining 30% of queries.

We observe that LimeQO reduces total latency to 0.9 hours (a 55% gain) while Greedy only reduces to 1.2 hours (a 40% gain). Additionally, LimeQO reaches the same performance level as when 100% of the queries were available from the start after only **0.5** hours of processing the new queries. In contrast, Greedy takes longer than 4 hours to reach the performance level seen when all queries are available from the start.

This demonstrates that LimeQO adapts well to workload shifts while Greedy does not. It shows the advantages of LimeQO over simpler techniques, highlighting its robustness in dynamic environment.

5.4 Data Shift

Another important question we seek to answer is LimeQO's robustness under data drift. To evaluate this, we utilized two versions of the Stack dataset [38]: one from 2017 and another from 2019.

First, we analyze the similarities and differences between the two versions: the total default runtime increased from 1.16 hours to 1.46 hours, while the optimal runtime increased from 0.9 hours to 1.09 hours. Additionally, we examine whether the best hints for the

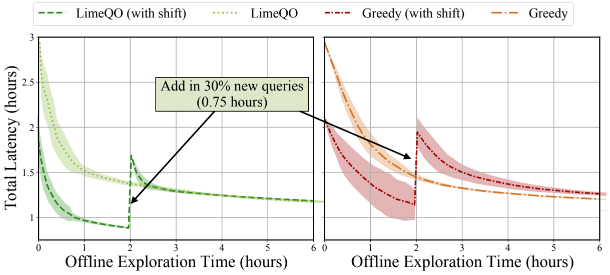

Figure 9: LimeQO (left figure) and Greedy (right figure) performance comparison when workload shift happens on the CEB workload. It shows LimeQO's ability to adapt to new queries.

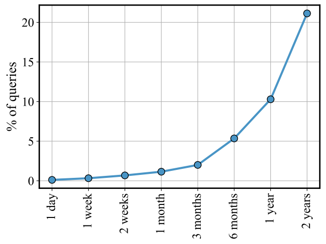

Figure 10: Incremental updates. The Y-axis is the % of queries with a different optimal hint. Note the non-uniform X-axis.

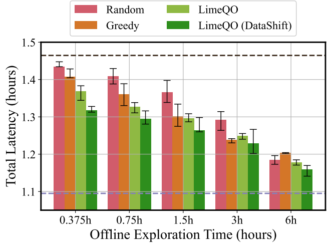

Figure 11: Data Shift on Stack workload showing LimeQO's ability to adapt to new data.

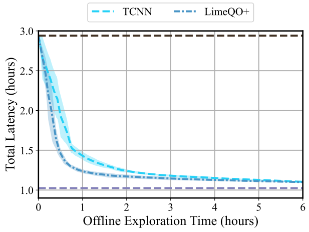

Figure 12: Total Latency comparison of TCNN VS. LimeQO+ on CEB workload.

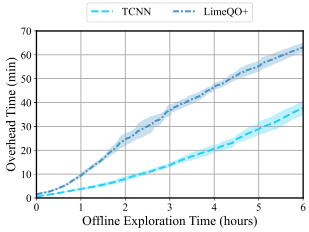

Figure 13: Overhead time comparison of TCNN VS. LimeQO+ on CEB workload.

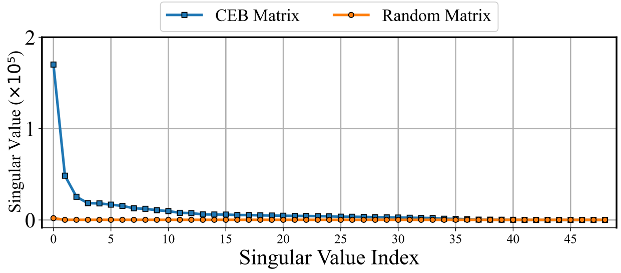

Figure 14: SVD on CEB workload matrix.

workload have changed and find that 79% of the workload queries maintain the same best hints. We further evaluated incremental updates using timestamp information, with intervals ranging from 1 day to 1 week, 1 month, and 1 year. As shown in Fig 10, updates with 1-day intervals result in negligible changes to the optimal hints. After 1 month, 1% of queries changed their optimal hints, 5% after 6 months, 10% after 1 year, and 21% after two years.

Applying the best hints from the 2017 dataset to the 2019 dataset reduced the total runtime from 1.46 hours to 1.26 hours, representing a 14% reduction compared to the optimal gain of 25%. In other words, even though the hints computed for the 2017 dataset are no longer optimal in 2019, the old hints *still improve latency by 14% compared to the default optimizer.*

Next, we simulate a complete data shift of two years after 4 hours of exploration. Given that 21% of workload queries changed the optimal hints after a 2 year data update - representing the maximum percentage observed in Fig 10 — this experiment reflects the worst-case impact of data shift. Specifically, we first explore the 2017 Stack dataset for 4 hours. Then, at the 4-hour mark, we entirely shift to the 2019 dataset. We begin exploring the new dataset using the current best hints derived from the previous dataset, and then continue with the same exploration process as described in Section 4.3.1. Figure 11 shows the total workload time after spending 0.25×, 0.5×, 1×, 2×, and 4× the default workload time (1.5 hours) on the 2019 new dataset. LimeQO is able to recover from the sudden data shift in thirty minutes, matching the performance of LimeQO when starting on the 2019 data.

5.5 Ablation Study

To better understand LimeQO and LimeQO+, we analyze: (1) the role of the TCNN component in LimeQO+; (2) the validity of the low-rank assumption; (3) the impact of rank selection on the performance; (4) the benefits of our censored techniques; and (5) our choice of ALS algorithm for matrix completion.

5.5.1 LimeQO+ vs. TCNN. As described in Section 4.3.2, LimeQO+ combines linear and neural methods. To evaluate the effectiveness of this integration, we compare the performance of the pure TCNN model with LimeQO+, noting that the TCNN component in both models is identical. Figure 12 shows that after 0.75 hours of optimization, LimeQO+ reduces latency by 56% from 2.9 hours to 1.3 hours, while TCNN only brings it down to 1.6 hours (a 48% reduction). It further shows that LimeQO+ consistently outperforms TCNN throughout the entire offline exploration process. This improvement can be attributed to the newly introduced features according to the low-rank property of the workload matrix, leading to more accurate predictions. We also compare the overhead time between TCNN and LimeQO+ in Figure 13. LimeQO+ spend about 20 additional minutes of overhead after exploring 6 hours. This confirms that the embedding layers in LimeQO+ improve performance significantly without introducing prohibitive overhead.

5.5.2 Low-rank Structure. A key assumption of LimeQO is that the workload matrix **W** has a low rank, without which matrix

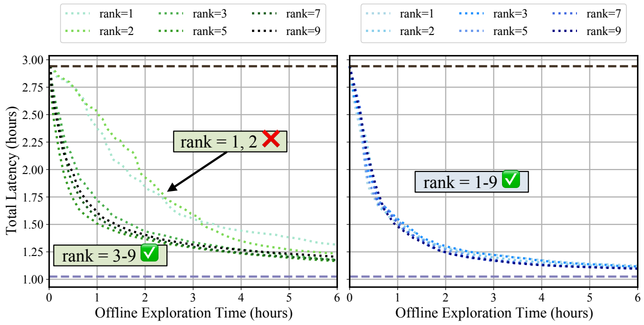

Figure 15: LimeQO (left figure) and LimeQO+ (right figure) performance on different ranks.

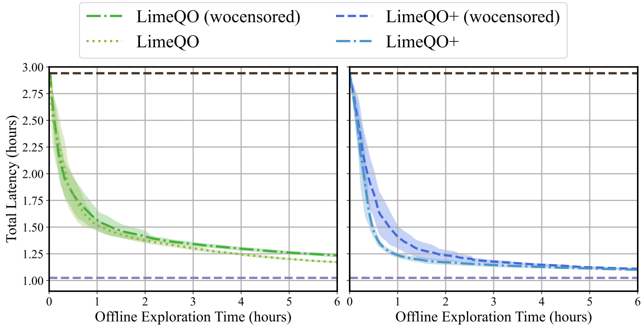

Figure 16: LimeQO (left figure) and LimeQO+ (right figure) performance with and without censored techniques.

completion may fail to predict unobserved plans accurately [8]. To validate this, we analyze the rank of the workload matrix for the CEB workload using singular value decomposition (SVD). Figure 14 presents the singular values of the complete **W** matrix, compared with those of a randomly generated matrix of the same shape. The workload matrix exhibits a few large singular values and many small ones, while the random matrix shows uniformly distributed singular values of similar magnitude. This observation confirms that the workload matrix can be well-approximated by a low-rank matrix, thus explaining why the ALS algorithm is effective in our scenario. For example, based on the singular values shown in Figure 14, we find that $r < 10$ is a reasonable choice, as it captures most of the significant information while discarding smaller, less impactful singular values.

5.5.3 Rank. Both LimeQO and LimeQO+ require an administrator to specify the rank r. To evaluate how sensitive the models are to this parameter, we experimented across a range of r. Figure 15 shows that LimeQO requires a rank greater than 2 to perform effectively, as ranks below this threshold fail to capture sufficient structure in the workload matrix. However, once the rank exceed two, the performance of LimeQO stabilizes and shows little variation. On the other hand, LimeQO+ demonstrates greater stability across different rank values, as it incorporates additional features from TCNN, making it less sensitive to changes in r. This observation aligns with the findings in Section 5.5.2 on singular values. Ultimately, we set $r = 5$, which offers a good balance between accurately approximating the workload matrix and maintaining computational efficiency.

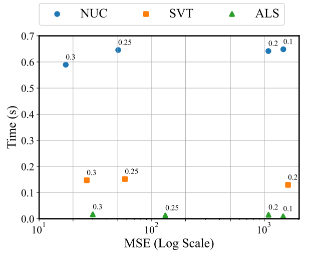

Figure 17: Comparison of Matrix Completion Techniques on the JOB workload matrix. The label next to each dot indicates the filled proportion p of the matrix.

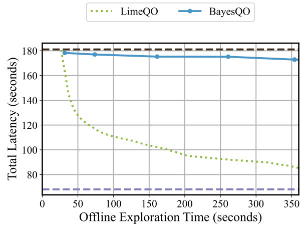

Figure 18: Comparison with BayesQO on the JOB workload. LimeQO significantly optimizes the workload, while BayesQO shows minimal progress.

5.5.4 Censored Techniques. We introduce censored techniques in both LimeQO and LimeQO+ to handle the time-out observations. Here, we evaluate the impact of these techniques on performance. In LimeQO, removing the censored technique is taking out lines 5 and 10 in Algorithm 2, thereby ignoring the timeout matrix. In LimeQO+, it entails training the model solely on non-censored data and relying the standard MSE loss function, which does not account for timeouts. Figure 16 shows that applying censored techniques to LimeQO results in less variance and improved performance after 2 hours of exploration. Similarly, LimeQO+ exhibits decreased variance and better performance. Specifically, LimeQO+ with censored technique reduces the total 3-hour workload to 1.5 hours after only 0.5 hours of exploration, whereas the version without censored technique took 0.9 hours to achieve the same reduction - a 1.8x longer exploration time. These results highlight the effectiveness of censored techniques in both methods, enabling them to achieve faster convergence and more consistent performance.

5.5.5 Comparisons of Matrix Completion Techniques. In this section, we explore three different matrix completion techniques and compare their accuracy and time overhead:

- **Nuclear Norm Minimization (NUC)** [9]: NUC recovers a low-rank matrix by minimizing the nuclear norm of the matrix subject to the constraints of observed entries, effectively leveraging the low rank property. While this approach can produce highly accurate results, it often requires substantial computational resources, especially for large datasets.
- **Singular Value Thresholding (SVT)** [7]: SVT uses singular value decomposition (SVD) and applies a threshold to the singular values to enforce low-rank approximation. However, it may struggle with noisy data or sparse observations.
- **Alternating Least Squares (ALS)** [21]: ALS iteratively optimizes matrix factors, makes it highly scalability. ALS is particularly effective for large-scale problems, as it can handle various forms of missing data and is less sensitive to initialization compared to other methods.

Figure 17 shows that while NUC provides good accuracy, it incurs a significant computational cost, taking over 0.5 seconds on the small JOB workload matrix (131 × 49). This overhead increases further for larger matrices. SVT, on the other hand, fails to handle sparse matrices effectively; its performance at $p = 0.1$ is absent because it could not solve the matrix under such sparsity. In contrast, ALS balances well between accuracy and efficiency, yielding satisfactory results with the least overhead across various levels of sparsity[4]. Therefore, we choose ALS as our matrix completion technique in Section 4.3.1.

5.6 Comparison with BayesQO

Our work shares similarities with BayesQO [67], a concurrent approach targeting offline query optimization. However, while BayesQO optimizes queries individually, our framework is designed to optimize an entire query workload simultaneously. To compare the two approaches, we conducted additional experiments using the JOB workload. Our method followed the approach described in Section 4.3.1. For BayesQO, each query in the workload was allocated a fixed optimization time of three seconds, after which the total latency was calculated. As shown in Figure 18, our approach achieves significant progress in optimizing the workload, whereas BayesQO barely makes progress on any single query. When optimizing an entire workload, it is advantageous to allocate exploration time dynamically to the "right" query, as opposed to allocating exploration time evenly among all queries.

6 Conclusions and Future Work

The question of how to effectively achieve the benefits of learned query optimization, *without* suffering performance regressions or running extensive model training, has been of strong interest to the database community. Building upon the idea of *offline* optimization, and using a method based on specifying *optimizer hints* to modify query optimizer behavior from an external system – this paper develops LimeQO: a framework for zero-regression, offline learned query optimization, without requiring extensive training, knowing specific plan features, or making assumptions about the underlying DBMS. Our methods are inspired by collaborative filtering, and are simple and low-overhead. Nonetheless, our experiments validate that, with appropriate active learning strategies, we can achieve nearly as much benefit as complex deep learning approaches. We also introduced LimeQO+, a more computationally expensive variant that integrates neural network techniques for faster convergence, but at the cost of higher overhead. Overall, LimeQO provides a practical solution for learned query optimization, ensuring efficient offline exploration without regressions and offering flexibility across different query optimization environments.

Future Work. Our existing framework relies on steering an optimizer through coarse-grained hints. In the future, we plan to investigate whether optimizers with finer-grained hints can benefit — for instance, the optimizer for the open-source Apache Presto [58] distributed SQL engine. Taking this one step further, we will explore whether modern optimizer frameworks such as Apache Calcite [5], used in many systems, could be extended to incorporate variations of our offline exploration model. We also plan to investigate techniques for *online* exploration over the space of hints and plans leveraging the low-rank structure, complementing the offline exploration of our current approach.

Additionally, since the query optimizer is PostgreSQL is less sophisticated than those found in commercial systems, we plan to extend our technique to systems like SQL Server and Oracle. On one hand, these commercial systems have more powerful baseline optimizers, so improvements may be harder to find. On the other hand, these commercial systems also provide a wider variety of hints, potentially creating more opportunities for optimization.

Evaluating our technique in the presense of data drift is also an important future direction. In this work, we looked at the impact of large data changes, but future work could evaluate performance on continuous data updates.

Finally, while we believe our incorporation of censored observations into ALS and TCNNs has been validated experimentally, future work could conduct a formal or theoretical analysis of censored techniques.

Acknowledgments

We thank the anonymous reviewers and our shepherd for their valuable feedback and suggestions.

References

[1] 2024. PostgreSQL database, http://www.postgresql.org/. http://www.postgresql.org/

[2] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. 1999. *LAPACK Users' Guide* (third ed.). Society for Industrial and Applied Mathematics, Philadelphia, PA.

[3] Christoph Anneser, Nesime Tatbul, David Cohen, Zhenggang Xu, Prithvi Pandian, Nikolay Leptev, and Ryan Marcus. 2023. AutoSteer: Learned Query Optimization for Any SQL Database. *PVLDB* 14, 1 (Aug. 2023). https://doi.org/10.14778/3611540.3611544

[4] Nikos Armenatzoglou, Sanuj Basu, Naga Bhanoori, Mengchu Cai, Naresh Chainani, Kiran Chinta, Venkatraman Govindaraju, Todd J. Green, Monish Gupta, Sebastian Hillig, Eric Hotinger, Yan Leshinksy, Jintian Liang, Michael McCreedy, Fabian Nagel, Ippokratis Pandis, Panos Parchas, Rahul Pathak, Orestis Polychroniou, Foyzur Rahman, Gaurav Saxena, Gokul Soundararajan, Sriram Subramanian, and Doug Terry. 2022. Amazon Redshift Re-invented. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 2205–2217. https://doi.org/10.1145/3514221.3526045

[5] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J Mior, and Daniel Lemire. 2018. Apache calcite: A foundational framework for optimized query processing over heterogeneous data sources. In *Proceedings of the 2018 International Conference on Management of Data*. 221–230.

[6] Djallel Bouneffouf, Romain Laroche, Tanguy Urvoy, Raphael Feraud, and Robin Allesiardo. 2014. Contextual Bandit for Active Learning: Active Thompson Sampling. In *Neural Information Processing (NeurIPS '14)*, Chu Kiong Loo, Keem Siah Yap, Kok Wai Wong, Andrew Teoh, and Kaizhu Huang (Eds.). Springer International Publishing, Cham, 405–412. https://doi.org/10.1007/978-3-319-12637-1_51

[7] Jian-Feng Cai, Emmanuel J. Candès, and Zuowei Shen. 2010. A Singular Value Thresholding Algorithm for Matrix Completion. *SIAM Journal on Optimization* 20, 4 (2010), 1956–1982. https://doi.org/10.1137/080738970 arXiv:https://doi.org/10.1137/080738970

[8] Emmanuel J. Candes and Terence Tao. 2009. The Power of Convex Relaxation: Near-Optimal Matrix Completion. http://arxiv.org/abs/0903.1476 arXiv:0903.1476 [cs, math].

[9] Emmanuel J. Candès and Benjamin Recht. 2009. Exact Matrix Completion via Convex Optimization. *Foundations of Computational Mathematics* 9, 6 (Dec. 2009), 717–772. https://doi.org/10.1007/s10208-009-9045-5

[10] Shayok Chakraborty, Jiayu Zhou, Vineeth Balasubramanian, Sethuraman Panchanathan, Ian Davidson, and Jieping Ye. 2013. Active Matrix Completion. In *2013 IEEE 13th International Conference on Data Mining*. IEEE, Dallas, TX, USA, 81–90. https://doi.org/10.1109/ICDM.2013.69

[4] We do not show $p > 0.3$ because we never observed a higher p in our experiments.

[11] Tianyi Chen, Jun Gao, Hedui Chen, and Yaofeng Tu. 2023. LOGER: A Learned Optimizer Towards Generating Efficient and Robust Query Execution Plans. *Proceedings of the VLDB Endowment* 16, 7 (March 2023), 1777–1789. https://doi.org/10.14778/3587136.3587150

[12] Xu Chen, Haitian Chen, Zibo Liang, Shuncheng Liu, Jinghong Wang, Kai Zeng, Han Su, and Kai Zheng. 2023. LEON: A New Framework for ML-Aided Query Optimization. *Proc. VLDB Endow.* 16, 9 (May 2023), 2261–2273. https://doi.org/10.14778/3598581.3598597

[13] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. 2017. Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 153–167. https://doi.org/10.1145/3132747.3132772

[14] Paul Covington, Jay Adams, and Emre Sargin. 2016. Deep Neural Networks for YouTube Recommendations. In *Proceedings of the 10th ACM Conference on Recommender Systems*. ACM, Boston Massachusetts USA, 191–198. https://doi.org/10.1145/2959100.2959190

[15] Guilherme Damasio, Vincent Corvinelli, Parke Godfrey, Piotr Mierzejewski, Alex Mihaylov, Jaroslaw Szlichta, and Calisto Zuzarte. 2019. Guided automated learning for query workload re-optimization. *Proceedings of the VLDB Endowment* 12, 12 (Aug. 2019), 2010–2021. https://doi.org/10.14778/3352063.3352120

[16] Bailu Ding, Surajit Chaudhuri, Johannes Gehrke, and Vivek Narasayya. 2021. DSB: A decision support benchmark for workload-driven and traditional database systems. *Proceedings of the VLDB Endowment* 14, 13 (2021), 3376–3388.

[17] Ravi Ganti and Alexander G. Gray. 2013. Building Bridges: Viewing Active Learning from the Multi-Armed Bandit Lens. https://doi.org/10.48550/arXiv.1309.6830 arXiv:1309.6830.

[18] Damjan Gjurovski, Angjela Davitkova, and Sebastian Michel. 2024. Grid-AR: A Grid-based Booster for Learned Cardinality Estimation and Range Joins. https://doi.org/10.48550/arXiv.2410.07895 arXiv:2410.07895 [cs].

[19] David Goldberg, David Nichols, Brian M Oki, and Douglas Terry. 1992. Using collaborative filtering to weave an information tapestry. *Commun. ACM* 35, 12 (1992), 61–70.

[20] Huifeng Guo, Ruiming Tang, Yunming Ye, Zhenguo Li, and Xiuqiang He. 2017. DeepFM: A Factorization-Machine based Neural Network for CTR Prediction. http://arxiv.org/abs/1703.04247 arXiv:1703.04247 [cs].

[21] Trevor Hastie, Rahul Mazumder, Jason Lee, and Reza Zadeh. 2014. Matrix Completion and Low-Rank SVD via Fast Alternating Least Squares. arXiv:1410.2596 [stat.ME]

[22] Xiangnan He, Lizi Liao, Hanwang Zhang, Liqiang Nie, Xia Hu, and Tat-Seng Chua. 2017. Neural Collaborative Filtering. http://arxiv.org/abs/1708.05031 arXiv:1708.05031 [cs].

[23] Benjamin Hilprecht and Carsten Binnig. 2022. Zero-shot cost models for out-of-the-box learned cost prediction. *Proceedings of the VLDB Endowment* 15, 11 (July 2022), 2361–2374. https://doi.org/10.14778/3551793.3551799

[24] Wei-Ning Hsu and Hsuan-Tien Lin. 2015. Active Learning by Learning. *Proceedings of the AAAI Conference on Artificial Intelligence* 29, 1 (Feb. 2015). https://doi.org/10.1609/aaai.v29i1.9597 Number: 1.

[25] Amin Kamali, Verena Kantere, Calisto Zuzarte, and Vincent Corvinelli. 2024. RobOpt: A Tool for Robust Workload Optimization Based on Uncertainty-Aware Machine Learning. In *Companion of the 2024 International Conference on Management of Data (SIGMOD '24)*. Association for Computing Machinery, New York, NY, USA, 468–471. https://doi.org/10.1145/3626246.3654755

[26] Amin Kamali, Verena Kantere, Calisto Zuzarte, and Vincent Corvinelli. 2024. Roq: Robust Query Optimization Based on a Risk-aware Learned Cost Model. (2024). https://doi.org/10.48550/ARXIV.2401.15210

[27] Ashish Kapoor, Eric Horvitz, and Sumit Basu. 2007. Selective supervision: Guiding supervised learning with decision-theoretic active learning. In *IJCAI'07 Proceedings of the 20th international joint conference on Artifical intelligence*. 877–882.

[28] Kyoungmin Kim, Sangoh Lee, Injung Kim, and Wook-Shin Han. 2024. ASM: Harmonizing Autoregressive Model, Sampling, and Multi-dimensional Statistics Merging for Cardinality Estimation. *Proc. ACM Manag. Data* 2, 1, Article 45 (March 2024), 27 pages. https://doi.org/10.1145/3639300

[29] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *3rd International Conference for Learning Representations (ICLR '15)*. San Diego, CA. https://arxiv.org/abs/1412.6980

[30] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter Boncz, and Alfons Kemper. 2019. Learned Cardinalities: Estimating Correlated Joins with Deep Learning. In *9th Biennial Conference on Innovative Data Systems Research (CIDR '19)*. http://arxiv.org/abs/1809.00677

[31] Yehuda Koren, Robert Bell, and Chris Volinsky. 2009. Matrix Factorization Techniques for Recommender Systems. *Computer* 42, 8 (Aug. 2009), 30–37. https://doi.org/10.1109/MC.2009.263 Conference Name: Computer.

[32] Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph Hellerstein, and Ion Stoica. 2018. Learning to Optimize Join Queries With Deep Reinforcement Learning. *arXiv:1808.03196 [cs]* (Aug. 2018). http://arxiv.org/abs/1808.03196 arXiv: 1808.03196.

[33] Yann LeCun and Yoshua Bengio. 1998. Convolutional networks for images, speech, and time series. *The Handbook of Brain Theory and Neural Networks* (1998), 255–258. http://dl.acm.org/citation.cfm?id=303568.303704

[34] Claude Lehmann, Pavel Sulimov, and Kurt Stockinger. 2024. Is Your Learned Query Optimizer Behaving As You Expect? A Machine Learning Perspective. *Proc. VLDB Endow.* 17, 7 (May 2024), 1565–1577. https://doi.org/10.14778/3654621.3654625

[35] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *PVLDB* 9, 3 (2015), 204–215. https://doi.org/10.14778/2850583.2850594

[36] Pengfei Li, Wenqing Wei, Rong Zhu, Bolin Ding, Jingren Zhou, and Hua Lu. 2023. ALECE: An Attention-based Learned Cardinality Estimator for SPJ Queries on Dynamic Workloads. *Proc. VLDB Endow.* 17, 2 (Oct. 2023), 197–210. https://doi.org/10.14778/3626292.3626302

[37] Wan Shen Lim, Lin Ma, William Zhang, Matthew Butrovich, Samuel Arch, and Andrew Pavlo. 2024. Hit the Gym: Accelerating Query Execution to Efficiently Bootstrap Behavior Models for Self-Driving Database Management Systems. *Proceedings of the VLDB Endowment* 17, 11 (July 2024), 3680–3693. https://doi.org/10.14778/3681954.3682030

[38] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2021. Bao: Making Learned Query Optimization Practical. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21)*. China. https://doi.org/10.1145/3448016.3452838 Award: 'best paper award'.

[39] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: A Learned Query Optimizer. *PVLDB* 12, 11 (2019), 1705–1718. https://doi.org/10.14778/3342263.3342644

[40] Ryan Marcus and Olga Papaemmanouil. 2018. Deep Reinforcement Learning for Join Order Enumeration. In *First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management (aiDM @ SIGMOD '18)*. Houston, TX.

[41] 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 (AAAI '16)*. AAAI Press, Phoenix, Arizona, 1287–1293. http://dl.acm.org/citation.cfm?id=3015812.3016002

[42] Raghunath Othayoth Nambiar and Meikel Poess. 2006. The Making of TPC-DS. In *VLDB (VLDB '06)*. VLDB Endowment, Seoul, Korea, 1049–1058. http://dl.acm.org/citation.cfm?id=1182635.1164217

[43] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Ilia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. 2019. Deep Learning Recommendation Model for Personalization and Recommendation Systems. http://arxiv.org/abs/1906.00091 arXiv:1906.00091 [cs].

[44] Parimarjan Negi, Matteo Interlandi, Ryan Marcus, Mohammad Alizadeh, Tim Kraska, Marc Friedman, and Alekh Jindal. 2021. Steering Query Optimizers: A Practical Take on Big Data Workloads. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21)*. ACM, Virtual Event China, 2557–2569. https://doi.org/10.1145/3448016.3457568 Award: 'best paper honorable mention'.

[45] Parimarjan Negi, Ryan Marcus, Andreas Kipf, Hongzi Mao, Nesime Tatbul, Tim Kraska, and Mohammad Alizadeh. 2021. Flow-loss: Learning cardinality estimates that matter. *Proc. VLDB Endow.* 14, 11 (2021), 2019–2032. https://doi.org/10.14778/3476249.3476259

[46] Parimarjan Negi, Ziniu Wu, Andreas Kipf, Nesime Tatbul, Ryan Marcus, Sam Madden, Tim Kraska, and Mohammad Alizadeh. 2023. Robust Query Driven Cardinality Estimation under Changing Workloads. *Proceedings of the VLDB Endowment* 16, 6 (April 2023), 1520–1533. https://doi.org/10.14778/3583140.3583164

[47] Chapelle Olivier, Schölkopf Bernhard, and Zien Alexander. 2006. 473A Discussion of Semi-Supervised Learning and Transduction. In *Semi-Supervised Learning*. The MIT Press. https://doi.org/10.7551/mitpress/9780262033589.003.0025 arXiv:https://academic.oup.com/mit-press-scholarship-online/book/0/chapter/353095099/chapter-ag-pdf/44419216/book_41571_section_353095099.ag.pdf

[48] Jennifer Ortiz, Magdalena Balazinska, Johannes Gehrke, and S. Sathiya Keerthi. 2018. Learning State Representations for Query Optimization with Deep Reinforcement Learning. In *2nd Workshop on Data Managmeent for End-to-End Machine Learning (DEEM '18)*. https://arxiv.org/abs/1803.08604

[49] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. In *Neural information processing workshops (NIPS-W '17)*.

[50] PostgreSQL Developers. 2024. PostgreSQL hints, https://www.postgresql.org/docs/current/runtime-config-query.html.

https://www.postgresql.org/docs/current/runtime-config-query.html tex.key= 1.

[51] Andy Ramlatchan, Mengyun Yang, Quan Liu, Min Li, Jianxin Wang, and Yaohang Li. 2018. A survey of matrix completion methods for recommendation systems. *Big Data Mining and Analytics* 1, 4 (Dec. 2018), 308–323. https://doi.org/10.26599/BDMA.2018.9020008 Conference Name: Big Data Mining and Analytics.

[52] Silvan Reiner and Michael Grossniklaus. 2024. Sample-Efficient Cardinality Estimation Using Geometric Deep Learning. *Proc. VLDB Endow.* 17, 4 (March 2024), 740–752. https://doi.org/10.14778/3636218.3636229

[53] Pengzhen Ren, Yun Xiao, Xiaojun Chang, Po-Yao Huang, Zhihui Li, Brij B. Gupta, Xiaojiang Chen, and Xin Wang. 2021. A Survey of Deep Active Learning. *ACM Comput. Surv.* 54, 9 (Oct. 2021), 180:1–180:40. https://doi.org/10.1145/3472291

[54] Steffen Rendle. 2010. Factorization Machines. In *2010 IEEE International Conference on Data Mining*. IEEE, Sydney, Australia, 995–1000. https://doi.org/10.1109/ICDM.2010.127

[55] Natali Ruchansky, Mark Crovella, and Evimaria Terzi. 2015. Matrix completion with queries. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 1025–1034. https://doi.org/10.1145/2783258.2783259 arXiv:1705.00399 [cs].

[56] Tobias Schmidt, Andreas Kipf, Dominik Horn, Gaurav Saxena, and Tim Kraska. 2024. Predicate Caching: Query-Driven Secondary Indexing for Cloud Data Warehouses. In *Companion of the 2024 International Conference on Management of Data*. ACM, Santiago AA Chile, 347–359. https://doi.org/10.1145/3626246.3653395

[57] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. 1979. Access Path Selection in a Relational Database Management System. In *SIGMOD '79 (SIGMOD '79)*, John Mylopolous and Michael Brodie (Eds.). Morgan Kaufmann, San Francisco (CA), 511–522. https://doi.org/10.1016/B978-0-934613-53-8.50038-8

[58] Raghav Sethi, Martin Traverso, Dain Sundstrom, David Phillips, Wenlei Xie, Yutian Sun, Nezih Yegitbasi, Haozhun Jin, Eric Hwang, Nileema Shingte, and Christopher Berner. 2019. Presto: SQL on Everything. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. 1802–1813. https://doi.org/10.1109/ICDE.2019.00196

[59] Burr Settles. 2009. Active learning literature survey. (2009).

[60] Jorge Silva and Lawrence Carin. 2012. Active learning for online bayesian matrix factorization. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*. 325–333.

[61] Nathan Srebro, Jason Rennie, and Tommi Jaakkola. 2004. Maximum-Margin Matrix Factorization. In *Advances in Neural Information Processing Systems*, Vol. 17. MIT Press. https://papers.nips.cc/paper_files/paper/2004/hash/e0688d13958a19e087e123148555e4b4-Abstract.html

[62] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research* 15, 1 (Jan. 2014), 1929–1958.

[63] Michael Stillger, Guy M. Lohman, Volker Markl, and Mokhtar Kandil. 2001. LEO - DB2's LEarning Optimizer. In *VLDB (VLDB '01)*. 19–28. http://dl.acm.org/citation.cfm?id=645927.672349

[64] Yutian Sun, Tim Meehan, Rebecca Schlussel, Wenlei Xie, Masha Basmanova, Orri Erling, Andrii Rosa, Shixuan Fan, Rongrong Zhong, Arun Thirupathi, Nikhil Collooru, Ke Wang, Sameer Agarwal, Arjun Gupta, Dionysios Logothetis, Kostas Xirogiannopoulos, Amit Dutta, Varun Gajjala, Rohit Jain, Ajay Palakuzhy, Prithvi Pandian, Sergey Pershin, Abhisek Saikia, Pranjal Shankhdhar, Neerad Somanchi, Swapnil Tailor, Jialiang Tan, Sreeni Viswanadha, Zac Wen, Biswapesh Chattopadhyay, Bin Fan, Deepak Majeti, and Aditi Pandit. 2023. Presto: A Decade of SQL Analytics at Meta. *Proceedings of the ACM on Management of Data* 1, 2 (June 2023), 189:1–189:25. https://doi.org/10.1145/3589769

[65] Danica J. Sutherland, Barnabás Póczos, and Jeff Schneider. 2013. Active learning and search on low-rank matrices. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, Chicago Illinois USA, 212–220. https://doi.org/10.1145/2487575.2487627

[66] Ki Hyun Tae, Hantian Zhang, Jaeyoung Park, Kexin Rong, and Steven Euijong Whang. 2024. Falcon: Fair Active Learning using Multi-armed Bandits. https://doi.org/10.48550/arXiv.2401.12722 arXiv:2401.12722.

[67] Jeffrey Tao, Natalie Maus, Haydn Jones, Yimeng Zeng, Jacob R. Gardner, and Ryan Marcus. 2025. Learned Offline Query Planning via Bayesian Optimization. arXiv:2502.05256 [cs.DB] https://arxiv.org/abs/2502.05256

[68] Robin Van De Water, Francesco Ventura, Zoi Kaoudi, Jorge-Arnulfo Quiané-Ruiz, and Volker Markl. 2022. Farming Your ML-based Query Optimizer's Food. In *2022 IEEE 38th International Conference on Data Engineering (ICDE) (ICDE '22)*. 3186–3189. https://doi.org/10.1109/ICDE53745.2022.00294 ISSN: 2375-026X.

[69] Alexander Van Renen, Dominik Horn, Pascal Pfeil, Kapil Vaidya, Wenjian Dong, Murali Narayanaswamy, Zhengchun Liu, Gaurav Saxena, Andreas Kipf, and Tim Kraska. 2024. Why TPC is Not Enough: An Analysis of the Amazon Redshift Fleet. *Proceedings of the VLDB Endowment* 17, 11 (July 2024), 3694–3706. https://doi.org/10.14778/3681954.3682031

[70] Alexander van Renen, Dominik Horn, Pascal Pfeil, Kapil Eknath Vaidya, Wenjian Dong, Murali Narayanaswamy, Zhengchun Liu, Gaurav Saxena, Andreas Kipf, and Tim Kraska. 2024. Why TPC is not enough: An analysis of the Amazon Redshift fleet. *Proceedings of the VLDB Endowment* (2024). https://www.amazon.science/publications/why-tpc-is-not-enough-an-analysis-of-the-amazon-redshift-fleet

[71] Ping Wang, Yan Li, and Chandan K. Reddy. 2019. Machine Learning for Survival Analysis: A Survey. *ACM Comput. Surv.* 51, 6 (Feb. 2019), 110:1–110:36. https://doi.org/10.1145/3214306

[72] Lianggui Weng, Rong Zhu, Di Wu, Bolin Ding, Bolong Zheng, and Jingren Zhou. 2024. Eraser: Eliminating Performance Regression on Learned Query Optimizer. *PVLDB* 17, 5 (2024), 926–938. https://doi.org/10.14778/3641204.3641205

[73] Lucas Woltmann, Jerome Thiessat, Claudio Hartmann, Dirk Habich, and Wolfgang Lehner. 2023. FASTgres: Making Learned Query Optimizer Hinting Effective. *Proceedings of the VLDB Endowment* 16, 11 (Aug. 2023), 3310–3322. https://doi.org/10.14778/3611479.3611528

[74] Peizhi Wu and Zachary G. Ives. 2024. Modeling Shifting Workloads for Learned Database Systems. *Proceedings of the ACM on Management of Data* 2, 1 (March 2024), 1–27. https://doi.org/10.1145/3639293

[75] Ziniu Wu, Ryan Marcus, Zhengchun Liu, Parimarjan Negi, Vikram Nathan, Pascal Pfeil, Gaurav Saxena, Mohammad Rahman, Balakrishnan Narayanaswamy, and Tim Kraska. 2024. Stage: Query Execution Time Prediction in Amazon Redshift. In *Proceedings of the 2024 International Conference on Management of Data (SIGMOD '24) (SIGMOD '24)*. Santiago, Chile. https://doi.org/10.48550/arXiv.2403.02286

[76] Zongheng Yang, Wei-Lin Chiang, Sifei Luan, Gautam Mittal, Michael Luo, and Ion Stoica. 2022. Balsa: Learning a Query Optimizer Without Expert Demonstrations. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 931–944. https://doi.org/10.1145/3514221.3517885

[77] Zongheng Yang, Eric Liang, Amog Kamsetty, Chenggang Wu, Yan Duan, Xi Chen, Pieter Abbeel, Joseph M. Hellerstein, Sanjay Krishnan, and Ion Stoica. 2019. Deep unsupervised cardinality estimation. *Proceedings of the VLDB Endowment* 13, 3 (Nov. 2019), 279–292. https://doi.org/10.14778/3368289.3368294

[78] Zixuan Yi, Yao Tian, Zachary G Ives, and Ryan Marcus. 2024. Low Rank Approximation for Learned Query Optimization. In *Proceedings of the Seventh International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*. 1–5.

[79] Xiang Yu, Chengliang Chai, Guoliang Li, and Jiabin Liu. 2022. Cost-Based or Learning-Based? A Hybrid Query Optimizer for Query Plan Selection. *Proceedings of the VLDB Endowment* 15, 13 (Sept. 2022), 3924–3936. https://doi.org/10.14778/3565838.3565846

[80] Xiang Yu, Guoliang Li, Chengliang Chai, and Nan Tang. 2020. Reinforcement Learning with Tree-LSTM for Join Order Selection. In *2020 IEEE 36th International Conference on Data Engineering (ICDE '20)*. 1297–1308. https://doi.org/10.1109/ICDE48307.2020.00116 ISSN: 2375-026X.

[81] Shuai Zhang, Lina Yao, Aixin Sun, and Yi Tay. 2020. Deep Learning based Recommender System: A Survey and New Perspectives. *Comput. Surveys* 52, 1 (Jan. 2020), 1–38. https://doi.org/10.1145/3285029 arXiv:1707.07435 [cs].

[82] Wangda Zhang, Matteo Interlandi, Paul Mineiro, Shi Qiao, Nasim Ghazanfari, Karlen Lie, Marc Friedman, Rafah Hosn, Hiren Patel, and Alekh Jindal. 2022. Deploying a Steered Query Optimizer in Production at Microsoft. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22)*. ACM, Philadelphia PA USA, 2299–2311. https://doi.org/10.1145/3514221.3526052

[83] William Zhang, Wan Shen Lim, Matthew Butrovich, and Andrew Pavlo. 2024. The Holon Approach for Simultaneously Tuning Multiple Components in a Self-Driving Database Management System with Machine Learning via Synthesized Proto-Actions. *Proceedings of the VLDB Endowment* 17, 11 (July 2024), 3373–3387. https://doi.org/10.14778/3681954.3682007

[84] Wei Zhou, Chen Lin, Xuanhe Zhou, and Guoliang Li. 2024. Breaking It Down: An In-Depth Study of Index Advisors. *Proceedings of the VLDB Endowment* 17, 10 (June 2024), 2405–2418. https://doi.org/10.14778/3675034.3675035

[85] Rong Zhu, Wei Chen, Bolin Ding, Xingguang Chen, Andreas Pfadler, Ziniu Wu, and Jingren Zhou. 2023. Lero: A Learning-to-Rank Query Optimizer. *Proceedings of the VLDB Endowment* 16, 6 (Feb. 2023), 1466–1479. https://doi.org/10.14778/3583140.3583160

[86] Rong Zhu, Lianggui Weng, Wenqing Wei, Di Wu, Jiazhen Peng, Yifan Wang, Bolin Ding, Defu Lian, Bolong Zheng, and Jingren Zhou. 2024. PilotScope: Steering Databases with Machine Learning Drivers. *PVLDB* 17, 5 (2024), 980–993. https://doi.org/10.14778/3641204.3641209

Received October 2024; revised January 2025; accepted February 2025