Low Rank Approximation for Learned Query Optimization

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

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

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

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

ABSTRACT

We present LimeQO, a learned steering query optimizer based on linear methods, such as matrix completion, for repetitive workloads. LimeQO can forgo expensive neural networks by taking advantage of the low-rank structure of query workloads. Using offline execution, LimeQO can accelerate workloads by up to 2x with zero regressions in just a few hours, while using 100-1000x fewer computational resources than deep learning techniques.

ACM Reference Format:
Zixuan Yi, Yao Tian, Zachary G. Ives, and Ryan Marcus. 2024. Low Rank Approximation for Learned Query Optimization. In *Seventh International Workshop on Exploiting Artificial Intelligence Techniques for Data Management (aiDM '24), June 14, 2024, Santiago, AA, Chile.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3663742.3663974

1 INTRODUCTION

Recent advances in learned query optimization — using machine learning to completely replace or aid a traditional query optimizer [20] — have demonstrated significant performance gains [13, 26–28, 31]. 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* (e.g., from neural networks [14] or from training data collection times [26]), and (3) they often *make assumptions about the underlying DBMS*, such as the availability of features or the structure of query plans.

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 [2, 30]. 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 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.

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 [2, 30]. While our framework can take advantage of expensive neural networks, we introduce a new approach, called LimeQO, which is trained with *purely **li**near **me**thods*, leading to drastically simpler and lower overhead implementations. Furthermore, LimeQO makes no assumptions about the underlying DBMS, except that each query plan has a number of alternatives with measurable latency: we do not assume the presence of a cardinality estimator, cost model, or even an operator tree.

Our core insight is to model the problem of offline optimization as a matrix completion [8] problem (a technique previously used for collaborative filtering). We can represent a workload with n repeated queries and k hints (parameterizations of the query optimizer that potentially result in different query plans, as in [2, 13, 26, 30]) 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, so 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).

One way to predict the missing entries in $\mathbf{W}$ is to train a predictive model on the observed entries of $\mathbf{W}$, then use that predictive

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

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

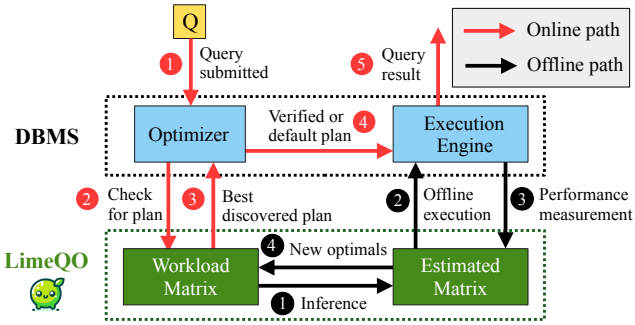

Figure 2: LimeQO system model

aiDM '24, June 14, 2024, Santiago, AA, Chile

ACM ISBN 979-8-4007-0680-6/24/06…$15.00
https://doi.org/10.1145/3663742.3663974

model to construct an estimate $\hat{\mathbf{W}}$. Unsurprisingly, computationally expensive tree convolution neural networks (TCNNs) [14, 16] can do a good job of approximating $\mathbf{W}$. Surprisingly, however, we find that the workload matrix $\mathbf{W}$ has a low rank r, which means that (among other things) we can construct an accurate estimate using two factored matrices $\mathbf{A} \in \mathbb{R}^{n \times r}$ and $\mathbf{B} \in \mathbb{R}^{k \times r}$: $\hat{\mathbf{W}} = \mathbf{A}\mathbf{B}^T$. This approximation can be found using purely linear methods that use **100x less computational resources than TCNNs,** and reduces inference time to the dot product of two r-dimensional vectors. Intuitively, $\mathbf{W}$ is low rank because *two queries that behave similarly on some hints are likely to behave similarly on other hints as well*; this means that a large portion of $\mathbf{W}$ can be explained by commonalities between queries and hints in the user's specific workload.

Experimentally, we show that LimeQO can accelerate a 10-hour workload **by a factor of 2 with only a few hours of offline exploration time**, making LimeQO's performance comparable to expensive neural networks. We believe that our initial results demonstrate that linear methods are promising additions to a suite of learned optimization techniques. In ongoing work, we are considering several open challenges, such as incorporating novel queries that were not part of the training set, as well as integrating linear and neural estimation techniques (Section 6).

We make the following contributions:

- We formalize the problem of offline exploration for query optimization in Section 2.
- We present LimeQO, a learned query optimizer that uses only **li**near **me**thods in Section 3.
- We present a preliminary experimental evaluation of LimeQO in Section 4.

2 PROBLEM DEFINITION

Our system model is depicted in Figure 2. 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 [6], 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. ❸ These promising plans are executed, ❹ and their performance is recorded. ❺ The now-observed values are then stored in the workload matrix.

A naive implementation of LimeQO could simply evaluate random unobserved query plans, but this strategy could waste offline execution time testing bad plans. Thus, LimeQO must strategically use each moment of offline execution time to create the largest improvement to the overall workload.

Next, we formalize this problem.

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 computing $\mathbf{W}$ is prohibitive, 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 $\tilde{\mathbf{W}}$:

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

Our goal is to simultaneously minimize the workload latency $P(\tilde{\mathbf{W}})$ and the total offline exploration time $T(\tilde{\mathbf{W}})$. Note that $P(\tilde{\mathbf{W}})$ can be trivially independently minimized by exploring all of $\mathbf{W}$, while $T(\tilde{\mathbf{W}})$ can be trivially independently minimized by doing no exploration at all. Thus, our goal is to find an algorithm that minimizes both simultaneously.

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 [2, 30]. A recent study 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 [33]. Thus, we consider repeated workloads an acceptable target for this preliminary work. We discuss extensions to novel queries in Section 6.

3 OFFLINE EXPLORATION

We propose two greedy techniques: linear methods based on matrix completion [8] (Section 3.1) and neural methods based on tree convolution neural networks (TCNN) [14, 16] (Section 3.2).

3.1 Linear methods

At a high level, LimeQO works by approximating $\mathbf{W}$ using *matrix completion* [8]: by assuming that $\mathbf{W}$ has low rank, the observed entries of $\mathbf{W}$ can be used to constrain the unobserved entries. After LimeQO constructs an approximation $\hat{\mathbf{W}}$ of $\mathbf{W}$, LimeQO selects the query plan with the largest expected benefit to explore. 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).

[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.

Algorithm 1: ALS

Input: $\tilde{\mathbf{W}}$: observed matrix; $\mathbf{M}$: mask matrix; k: rank; λ: regularization parameter; t: number of iterations
Output: Completed matrix $\hat{\mathbf{W}}$

1 Initialize A, B of size n × r, and k × r randomly ;
2 **for** $i = 1$ *to* t **do**
3 $\quad \hat{\mathbf{W}} \leftarrow \mathbf{M} \odot \tilde{\mathbf{W}} + (1 - \mathbf{M}) \odot AB^T$;
4 $\quad A \leftarrow \hat{\mathbf{W}}B(B^TB + \lambda I)^{-1}$;
5 $\quad A[A < 0] = 0$;
6 $\quad \hat{\mathbf{W}} \leftarrow \mathbf{M} \odot \tilde{\mathbf{W}} + (1 - \mathbf{M}) \odot AB^T$;
7 $\quad B \leftarrow \hat{\mathbf{W}}A(A^TA + \lambda I)^{-1}$;
8 $\quad B[B < 0] = 0$;
9 $\hat{\mathbf{W}} \leftarrow \mathbf{M} \odot \tilde{\mathbf{W}} + (1 - \mathbf{M}) \odot AB^T$;
10 **return** $\hat{\mathbf{W}}$

Matrix completion. Matrix completion (MC) is a technique used to recover unobserved entries in a low rank matrix [3, 4, 8, 21]. 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 an estimate of $\mathbf{W}$ as $\hat{\mathbf{W}} = \mathbf{AB}^T$ by solving:

$$\min_{\mathbf{A},\mathbf{B}} \left[\|\mathbf{M} \odot (\tilde{\mathbf{W}} - \mathbf{AB}^T)\|_F^2 + \lambda \left(\|\mathbf{A}\|_F^2 + \|\mathbf{B}\|_F^2 \right) \right] \tag{4}$$

where $\mathbf{A}$ and $\mathbf{B}$ are $n \times r$ and $k \times r$ matrices, respectively, and $\odot$ represents the element-wise product. To find $\mathbf{A}$ and $\mathbf{B}$, we use the alternating least squares (ALS) algorithm [8], which is based on the following key observation: while Equation 4 is not convex in $\mathbf{A}$ and $\mathbf{B}$, Equation 4 is convex in $\mathbf{A}$ for fixed $\mathbf{B}$, and vice versa. Thus, ALS works by alternating between fitting $\mathbf{A}$ assuming a fixed $\mathbf{B}$, and then fitting $\mathbf{B}$ assuming a fixed $\mathbf{A}$ (Algorithm 1). Since query latencies are strictly positive, we additionally constrain $\mathbf{A}$ and $\mathbf{B}$ to be non-negative after each step (lines 5 and 8).

Using MC. Next, we explain how LimeQO uses MC for query optimization, summarized in Algorithm 2. Given an initial $\tilde{\mathbf{W}}$, we first find the current best hint for each query (Line 2-3). Then, we use MC to construct an estimate $\hat{\mathbf{W}}$ (Line 4). With the estimated value, we go through every row of the predicted matrix and compute *potential improvement*, which is the difference between the best observed plan and the predicted best plan for a query. Then, we sort the improvements and select the top m entries (Line 9). In the case where there are less than m positive predicted improvements (Line 10), we will randomly select some unobserved entries (Line 11) to observe. Finally, we execute the m selected plans, record their latency, and update $\mathbf{M}$ 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.

3.2 Neural methods

Here, we present an alternative approach to solve offline query optimization using a TCNN. Our approach is similar to [13, 14], and we assume query plan features are available (e.g., cost estimates), and that the underlying optimizer generates tree-structured plans.

Algorithm 2: LimeQO

Input: $\tilde{\mathbf{W}}$: initial observed matrix; $\mathbf{M}$: mask matrix; k: rank; λ: regularization parameter; t: number of iterations
Output: Hint selections $[h_1, \ldots, h_n]$ for workload

1 **while** $\mathbf{M} \neq \mathbf{1}$ **do**
2 $\quad$ **for** $i = 1$ *to* n **do**
3 $\quad\quad h_i \leftarrow H[\text{argmin}_j(\tilde{\mathbf{W}}_{ij})]$;
4 $\quad \hat{\mathbf{W}} \leftarrow$ ALS $(\tilde{\mathbf{W}}, \mathbf{M}, k, \lambda, t)$;
5 $\quad$ **for** $i = 1$ *to* n **do**
6 $\quad\quad h_j \leftarrow H[\text{argmin}_j(\hat{\mathbf{W}}_{ij})]$;
7 $\quad\quad \Delta\mathbf{W}_i \leftarrow \min(\tilde{\mathbf{W}}_i) - \hat{\mathbf{W}}_{ij}$;
8 $\quad\quad$ add (q_i, h_j) to S if $\Delta\mathbf{W}_{ij} > 0$;
9 $\quad$ Select top m largest (q_i, h_j) from S w.r.t. $\Delta\mathbf{W}_{ij}$;
10 $\quad$ **if** *not enough to select* **then**
11 $\quad\quad$ randomly select some unobserved (q_i, h_j);
12 $\quad$ Update $\mathbf{M}$ and $\tilde{\mathbf{W}}$;
13 **return** $[h_1, \ldots, h_n]$

Tree convolution. Tree convolution [16] is a neural network operator that slides tree-shaped filters over query plans, identifying patterns related to query performance [14]. As a result, TCNNs are widely used for query optimization problems [2, 9, 13, 27].

Adapting TCNN to our task. Using a TCNN for offline exploration only requires a small change to Algorithm 2: instead of using MC to construct an estimate of the workload matrix (Line 4), we train a TCNN model and use it to predict each unobserved value in $\tilde{\mathbf{W}}$. Specifically, the TCNN model is trained on the observed entries of $\tilde{\mathbf{W}}$, using plan features extracted from each query plan. Then, inference is conducted to generate predicted latencies for all unobserved entries. Consequently, $\hat{\mathbf{W}}$ consists of the actual latencies for observed entries and the TCNN's predictions for unobserved ones.

4 INITIAL EXPERIMENTAL RESULTS

Our preliminary experimental results seek to answer three key questions:

(1) How does LimeQO's performance compare to simple baselines and deep learning methods? (Section 4.1)
(2) How does the overhead of LimeQO's linear methods compare to the overhead of deep learning methods? (Section 4.2)
(3) What is the approximate rank of the workload matrix? (Section 4.3)

Experimental setup. We evaluated LimeQO using PostgreSQL 16.1 [1] and the CEB core workload (3133 queries) [17] over the IMDb database [12]. LimeQO uses the same 49 hints as Bao [13], 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.[2] All experiments were conducted on a server running 64-bit Ubuntu 22.04 with Intel(R) Xeon(R) Gold 6248R CPU @ 3.00GHz, 503 GB RAM, and an NVIDIA A100 GPU. Without LimeQO, PostgreSQL runs the CEB workload

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

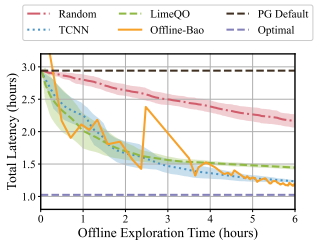

Figure 3: Total latency

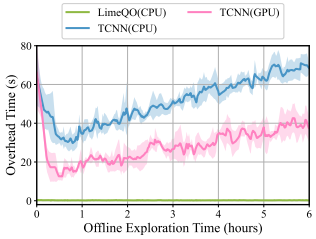

Figure 4: Overhead

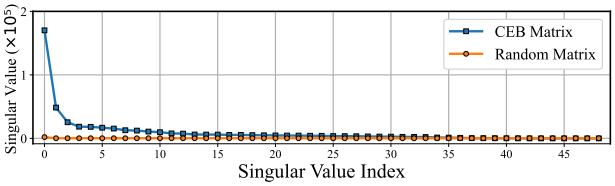

Figure 5: Singular Value Plot

in 3 hours. Each experiment is executed five times; we plot the average and standard deviation.

Techniques tests. We compare four 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.

- LimeQO: use MC to explore the matrix as described in Section 3.1. We set $r = 5$, $\lambda = 0.2$ and $t = 50$ in Algorithm 1.
- Random: explore the workload matrix by randomly selecting unobserved entries.
- TCNN: uses TCNN to predict unobserved matrix entries as described in Section 3.2. We use the same TCNN architecture as [13], except that we add a dropout layer [22] with $p = 0.3$ between each tree convolution layer, which universally improved results. Training is performed with Adam [10] 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.
- Offline-Bao: the technique of [13] adapted to offline exploration: the TCNN is used to select unobserved entries to explore, but the TCNN model is fully trusted in the online path (and thus regressions are possible).

4.1 Performance improvements

Figure 3 shows that with 2 hours of offline exploration time, LimeQO and TCNN can reduce the overall workload time to 1.7 hours (56% of the default workload execution). The relatively poor performance of Random indicates that this is not due to chance. Following a total exploration time of 6 hours, LimeQO reduces total latency to 1.5 hours, while TCNN reduces total latency to 1.25 hours. This represents a speed-up of 2x and 2.4x, respectively.

Our linear method (LimeQO) initially outperforms the neural method. TCNN later surpasses LimeQO after 1.5 hours of exploration time and achieves a lower total latency time. This shift can be attributed to TCNN's deep learning approach, which improves its performance as it receives more training data. However, this advantage comes at the cost of much larger training, inference time, and space overhead. Additionally, TCNN requires rich plan tree features, while LimeQO does not require such features.

Figure 3 also illustrates the variability in Offline-Bao: since Offline-Bao does not have to verify plans before selecting them, Offline-Bao can make improvements quickly. But this performance improvement comes at the cost of query regressions. In that sense, the gap between the Offline-Bao line and the other techniques can be interpreted as "the cost of zero regressions."

4.2 Overhead

Figure 4 shows the overhead required for LimeQO and TCNN (Offline-Bao's overhead is the same as TCNN). Clearly, TCNN requires significantly more resources than LimeQO. In each exploration step, TCNN must train a model on observed plan trees and then perform inference for each unobserved plan tree, while LimeQO only needs to complete the matrix. LimeQO's overhead remains approximately constant at 200ms, whereas TCNN's overhead on CPU ranges from 30 to 80 seconds. Although two minutes of overhead may or may not seem significant, we note that TCNN's overhead scales with the number of observed entries. Even with a GPU, TCNN still requires 10 to 70 seconds of overhead time. This highlights that linear methods use computational resources that are at least 100 times more efficient.

It is also worth noting that the implementation of a TCNN requires significant complexity and has a large software footprint (e.g., PyTorch [19]). On the other hand, LimeQO's implementation only requires near-universal linear algebra routines.

4.3 Low-rank structure

A critical assumption made by LimeQO is that the workload matrix $\mathbf{W}$ has low rank. If $\mathbf{W}$ does not have low rank, it is unlikely that matrix completion will make accurate predictions for unobserved plans [4]. Here, we verify that the workload matrix for CEB is indeed low rank using singular value decomposition. Figure 5 shows the singular values of the complete $\mathbf{W}$ matrix and, for comparison, a randomly generated matrix. The singular values of the workload matrix consist of a few large values and many small values, whereas the singular values of the random matrix are uniformly distributed and of similar magnitude. This observation confirms that our workload matrix can be well approximated by a low rank matrix, thus explaining why the ALS algorithm is effective in our scenario.

5 RELATED WORKS

Recent work on learned query optimization is broadly divided into two categories: "full" learned optimizers that synthesize entire query plans [5, 9, 11, 14, 15, 18, 27, 29, 31], and "steering" learned optimizers that sit on top of a traditional optimizer [13, 26, 28]. The latter "steering" approach has fewer degrees of freedom, but exhibits lower variation, leading to adoption in some production systems [2, 30, 32]. Since any performance variation can be harmful to downstream applications, offline execution is often used to verify performance improvements [2, 24, 30], although confidence-learning based approaches are also being developed [9, 25].

Matrix completion is a decades-old [7] technique that has mostly seen applications in recommendation systems [8], although MC has also been the subject of deep mathematical investigation [3]. Linear methods have also been used by learned cardinality estimators [23].

6 FUTURE DIRECTIONS & CONCLUSIONS

This preliminary work has presented a framework for zero-regression offline learned query optimization, and shown how simple, low-overhead linear methods can be nearly as effective as complex deep learning approaches, without requiring any plan features or making assumptions about the underlying DBMS. Several open questions and challenges remain to be investigated.

Transductive techniques. In ML terms, techniques such as TCNNs are called *inductive* because they learn a model from training inputs and labels and then predict labels for unseen test inputs. LimeQO, on the other hand, is an *transductive* technique, since training inputs, training labels, and test data are known in advance (only test labels are unknown). What might a transductive TCNN look like? Can MC and TCNN be combined in some way that results in a smaller TCNN? Or perhaps there is a middle ground between dead simple techniques like MC and more complex techniques like TCNN?

Online optimization. A major weakness of our preliminary version of LimeQO is handling novel queries. Adding a new, empty row to the workload matrix with no entries results in any arbitrary prediction satisfying Equation 4 (an under-constrained linear system). It may be possible to match novel queries to similar previously observed queries in a predictable way. But an even simpler approach could be to execute novel queries using the default optimizer first, then adding a populated row to the matrix afterward. We leave the investigation and evaluation of such techniques to future work.

REFERENCES

[1] [n. d.]. PostgreSQL Database, http://www.postgresql.org/. ([n. d.]).

[2] 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

[3] 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].

[4] 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

[5] 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

[6] 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

[7] 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.

[8] 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]

[9] 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

[10] 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.

[11] 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). arXiv:1808.03196 [cs]

[12] 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

[13] 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

[14] 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

[15] 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.

[16] 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.

[17] 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

[18] 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)*.

[19] 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)*.

[20] 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

[21] 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

[22] 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.

[23] Michael Stillger, Guy M. Lohman, Volker Markl, and Mokhtar Kandil. 2001. LEO - DB2's LEarning Optimizer. In *VLDB (VLDB '01)*. 19–28.

[24] 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

[25] 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

[26] 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

[27] 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

[28] 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

[29] 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

[30] 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

[31] 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

[32] 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

[33] 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