Inductive Matrix Completion with Embedding Memory and Bandit Exploration for Safe Hint Recommendation (Extended Abstract)

Raahim Lone
Independent
Al Khobar, Saudi Arabia
raahimlone@gmail.com

ABSTRACT

Learned query optimizers struggle to generalize, causing performance regression for a subset of queries. To address this, DataSwift is introduced, a hint-recommendation framework that integrates LLM-derived SQL embeddings, GNN-encoded plan representations, a similarity-threshold memory cache, and Thompson-sampling bandit exploration. Incoming queries are embedded to recall proven hints; a low-rank inductive matrix completion model predicts expected latency. Validated hints are cached, and the bandit down-weights any hints inducing slowdowns. On the combined JOB benchmarks, DataSwift incurs only a **0.7%** regression rate with zero catastrophic regressions and delivers a **1.4x** improvement on the 5% slowest queries. Thus, DataSwift provides performance gains without sacrificing safety.

VLDB Workshop Reference Format:
Raahim Lone. Inductive Matrix Completion with Embedding Memory and Bandit Exploration for Safe Hint Recommendation (Extended Abstract). VLDB 2025 Workshop: 6th International Workshop on Applied AI for Database Systems and Applications.

VLDB Workshop Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/Raahim-Lone/DataSwift.git.

1 INTRODUCTION

Machine learning enhanced query optimizers promise to automatically discover efficient execution plans by learning cost models from historical query feedback [9]. However, current learned optimizers exhibit *performance regressions*: a query that ran quickly under the traditional optimizer runs slower under the learned model. Such slowdowns arise for three primary reasons:

(1) **Distribution Shift:** When a learned cost model encounters queries or data patterns not seen during training, predictions accuracy degrades [12].
(2) **Cold-start queries:** Many existing predictors (including Redshift's built in estimator) fall back to a less accurate model/heuristic upon seeing an unseen query template [13].
(3) **Lack of uncertainty quantification:** Most learned optimizers output a plan without measuring confidence. A mispredicted cost can steer the optimizers towards a plan that is catastrophically worse than the default [6].

Empirical studies confirm that performance regression events, occur frequently in query workloads [3, 12]. To counteract this, a growing body of work has sought to mitigate regressions by supplementing learned cost models with fallback mechanisms:

- **Offline plan validation.** Systems like LimeQO [14, 15] and AutoSteer [4] execute candidate plans ahead of time to ensure no regressions. While effective on repetitive analytic workloads, approaches like these depend on query patterns repeating.
- **Uncertainty modelling.** Approaches like Lero frames cost estimation as a learning to rank problem [16] and RoQ [7] employs probabilistic cost predictors to capture variance. While these models can reduce the amount of performance regressions, they suffer from several drawbacks:
  (1) **Retraining overhead.** Adapting to workload shifts requires full retraining, which is impractical for dynamic production environments.
  (2) **Variance misestimation.** Probabilistic estimators often misestimate uncertainty for novel queries.
- **Retrieval augmented and bandit-based adaptation.** Recent work integrates nearest-neighbor recall [12] and bandit learning [9] to adapt online. LLM-derived embeddings have been shown to capture query semantics effectively [2] while Thompson-sampling bandit can balance exploration and exploitation. However, there are drawbacks:
  (1) **Indexing and lookup cost.** Embedding-based retrieval over large historical workloads can incur significant latency and memory overhead.
  (2) **Variance misestimation.** Exploration can select poorly-performing plans without explicit bounds on regret, making them unsuitable for latency-sensitive applications.
- **Delta-based filters.** PerfGuard uses a plan-delta meta-model to predict regressions during pre-production flighting [3], and Eraser applies two-stage runtime reliability checks to filter out low-confidence plans [12]. PerfGuard remains offline-only, risking live slowdowns. Eraser depends on heavily tuned hyperparameters, coarse clustering, and purely plan-level features, resulting in gaps when novel plan shapes arise.

Proceedings of the VLDB Endowment. ISSN 2150-8097.

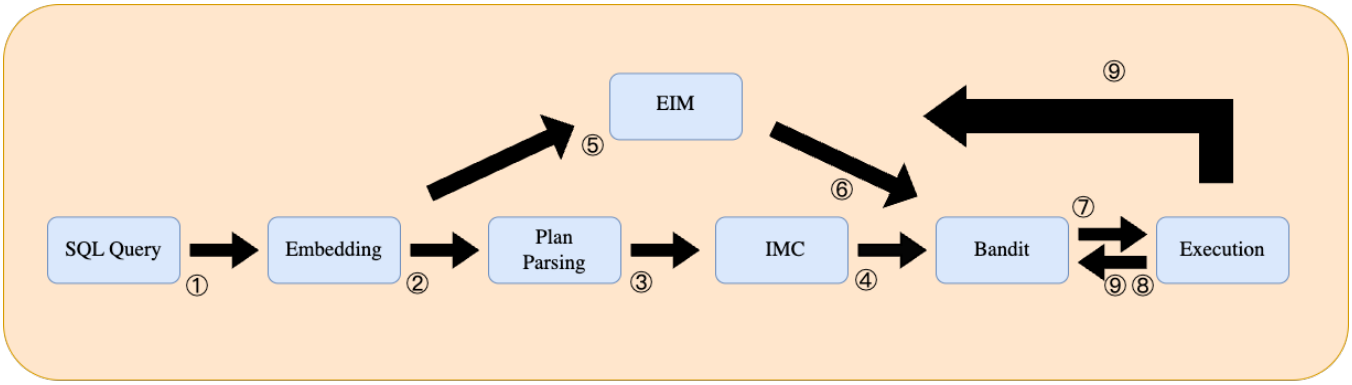


Figure 1: This figure demonstrates DataSwift, which takes incoming SQL queries, parses their plans, and uses an IMC model to predict hint performance. IMC training and EIM maintenance are conducted offline, while the rest of the workflow is conducted online.

Taken together, these studies demonstrate that a query-slowdown free, learned optimizer must fulfill three criteria:

(1) Expressive query representations to generalize to novel queries.
(2) A memory of past outcomes to avoid repeating mistakes on similar queries.
(3) Balanced exploration against exploitation under uncertainty without excessive overhead.

To address these needs, DataSwfit is proposed. The contributions are as follows:

(1) **LLM-derived query embeddings + GNN plan encodings.** Each SQL query is converted into a semantic embedding (via LLM) and a graph-based plan embedding (via GNN), enabling generalization to new queries.
(2) **Inductive matrix completion (IMC) with uncertainty estimation.** An IMC predictor takes the combined embedding and outputs variance estimates of latency for each hint, yielding uncertainty-aware ranking.
(3) **Similarity-threshold memory cache.** Mappings for past queries whose empirical performance is validated are stored, allowing retrieval of proven hints for similar queries.
(4) **Thompson-sampling bandit with safe fallback.** Each hint is treated as a bandit arm. By sampling from the IMC predictor, hints are selected and the IMC model and bandit are updated after execution. The bandit down-weights any hints that regress during exploration.

2 SYSTEM OVERVIEW

Upon receiving a raw SQL query (1), DataSwift simultaneously generates a fixed-length SQL embedding (2) using a pretrained SentenceTransformer, and parses the query into a logical operator DAG for a GNN-based plan embedding (3). These embeddings are concatenated and fed into the IMC predictor (4), which formulates hint selection as an IMC problem to output both a mean latency estimate and an uncertainty score for each candidate hint. Before applying the top IMC suggestion, DataSwift queries the Embedding-Indexed Memory (EIM) (5)-a vector index of past queries indexed by SQL embeddings—to identify any validated hint (6) as an additional candidate. Next, the Bandit Selector (7) treats the IMC's top suggestion, any EIM-returned hint, and the default (no-hint) option as arms in a multi-armed bandit. It samples from each arm's posterior to choose a hint. That chosen hint is then applied and sent to the execution engine (8). Finally, the observed execution latency updates the bandit's parameters and is stored back into EIM for future lookups (9).

2.1 SQL Embedding

The SQL embedding module transforms raw SQL text into a high-dimensional vector that captures semantic patterns. A Sentence-Transformer model pretrained and fine-tuned on large-scale SQL workloads is used. Each incoming query is tokenized and passed through the Transformer's encoder. During inference, the euclidean distance between the query's embedding and those stored in Embedding-Indexed Memory is computed. By relying on a pretrained LLM to capture deep semantic similarity, the variance estimates of purely probabilistic cost models are avoided. The need for offline validation on every new template is also removed, since proven hints can be recalled for highly similar queries.

2.2 Plan Graph encoding and IMC Prediction

Once the SQL text is embedded, the query's plan tree is parsed into a directed acyclic graph (DAG) and processed by a Graph Neural Network (GNN) to produce a 512-dimensional structural embedding $\mathbf{z}_{\text{struct}} \in \mathbb{R}^{512}$ [5]. A SentenceTransformer output is reduced via PCA to a 120-dimensional SQL embedding $\mathbf{z}_{\text{text}} \in \mathbb{R}^{120}$. The following is concatenated to allow for a unified feature vector:

$$\mathbf{x} = \begin{bmatrix} \mathbf{z}_{\text{struct}} \\ \mathbf{z}_{\text{text}} \end{bmatrix} \in \mathbb{R}^{632}.$$

This vector $\mathbf{x}$ is fed into the IMC module, which models a set of k candidate hints (the same 49 hints from Bao [9]). The following two low-rank projection matrices can be found:

$$U \in \mathbb{R}^{632 \times r}, \quad V \in \mathbb{R}^{d_h \times r},$$

Two scalar bias heads b_z, b_h can also be found. For each hint h with its own embedding $\mathbf{h}_h \in \mathbb{R}^{d_h}$, the following is computed:

$$\mathbf{u}_q = \mathbf{x}\,U, \quad \mathbf{v}_h = \mathbf{h}_h\,V, \quad \mu_{q,h} = \mathbf{u}_q^\top \mathbf{v}_h \;+\; b_z \;+\; b_h.$$

To recognize how confident the IMC predictiosn are, the following is formed:

$$\left[\mathbf{u}_q;\, \mathbf{v}_h;\, |\mathbf{u}_q - \mathbf{v}_h|\right] \;\in\; \mathbb{R}^{2r+1},$$

It is then passed through a small linear layer to yield an uncertainty score $\sigma_{q,h}$. During training, let

$$y_{q,h} = \log\bigl(1 + \ell_{q,h}\bigr), \quad \mathcal{I} = \bigl\{(q,h) \;\big|\; |y_{q,h} - \mu_{q,h}| < 2.5\,\sigma_{q,h}\bigr\}.$$

be established. Here, query–hint pairs whose predicted mean $\mu_{q,h}$ lies within 2.5 of its own standard deviation $\sigma_{q,h}$ from the true value $y_{q,h}$ are included. This avoids the model being dominated by outliers.

The formal IMC training objective [11] is adapted to minimize the negative log-likelihood:

$$\mathcal{L} = \frac{1}{|\mathcal{I}|} \sum_{(q,h)\in\mathcal{I}} \Bigl(\tfrac{1}{2}\log\sigma_{q,h} + \frac{(y_{q,h} - \mu_{q,h})^2}{2\,\sigma_{q,h}}\Bigr),$$

At inference time, for each hint h the pair $\bigl(\mu_{q,h},\ \sigma_{q,h}\bigr)$ is outputted and $1/\sigma_{q,h}$ is used as a confidence measure. The hint with the smallest predicted latency $\mu_{q,h}$ is selected. This IMC-with-uncertainty approach generalizes to new query-hint pairs without retraining. Furthermore, it imputes missing entries more flexibly than static plan-delta filters while still providing calibrated confidence for runtime decision-making.

2.3 Embedding-Indexed Memory (EIM)

The EIM component provides a case-based safety net by caching past queries together with their empirically validated optimal hints. An in-memory datastore of tuples

$$\{\,(\mathbf{e}_i, h_i^*, n_i, b_i)\},$$

where $\mathbf{e}_i$ is a *normalized* embedding of a previously executed query q_i, h_i^* is its best-observed hint, n_i is the number of executions, and b_i the number of "bad" trials.

At inference, a Faiss L_2 index (IndexFlatL2) on the stored embeddings is used. The top-k neighbors whose squared distance $\|\mathbf{e}_q - \mathbf{e}_i\|_2^2 \le \tau^2$ is retrieved. However, before selecting a hint, any neighbor that has failed in over 40 % of its past executions (once it has at least eight total runs) is discarded. After each query, the radius is adjusted $\tau \leftarrow \tau \times 0.92$ based on success or $\tau \leftarrow \tau \times 1.07$ on failure. A larger radius lets the model consider less similar queries and vice versa.

This mechanism addresses two scenarios: (i) **cold-start queries** that have no reliable IMC prediction (high uncertainty), and (ii) **recurring query patterns** where the IMC may have made an error. By recalling a proven hint for a highly similar query, the EIM reduces the likelihood of a query slowdown. The memory is periodically updated: after each query execution, if the chosen hint yields better latency than both the default and any previously stored hint for that embedding, $(\mathbf{e}_q, h_q)$ is updated. To bound memory size, a First-In-First-Out (FIFO) eviction policy is used.

By only invoking offline-validated hints when there's strong embedding similarity, EIM sidesteps the heavy execution overhead of no-regression guards. Likewise, EIM provides stronger safety than naive memory recall/bandit-only schemes because it anchors retrieval on proven outcomes.

2.4 Bandit-Based Hint Selection

After obtaining two potential hints—(i) the IMC's top recommendation (with its predicted latency and confidence) and (ii) any hint retrieved from the EIM—the final decision is made by a Thompson-sampling multi-armed bandit. Each candidate source is treated as an "arm":

- **Arm** A_{IMC}: the hint h_{IMC} with lowest latency from the IMC prediction, with an associated posterior parameterized by the IMC's uncertainty estimate $\sigma_{q,h}$.
- **Arm** A_{EIM}: if the EIM returns one or more hints $\{\,h_{\text{mem},i}\}$, each distinct hint is treated as a separate arm; otherwise, this arm is absent.
- **Arm** A_{DFLT}: the "no-hint" fallback/using the database's default optimizer.

The bandit maintains, for each arm a, a posterior distribution over its expected reward r_a.+ The reward is defined as the observed speedup

$$r_a = \frac{\ell_{\text{default}}}{\ell_{q,a}},$$

where ℓ_{default} is the latency observed under the default plan, and $\ell_{q,a}$ is the latency under hint a. A positive reward indicates a speedup over default and vice versa. Each execution provides a sample $\tilde{r}_a$ used to update the posterior of arm a. When selecting an arm, Thompson sampling draws a random sample from each arm's posterior. Then, it chooses the arm with the highest sampled reward.

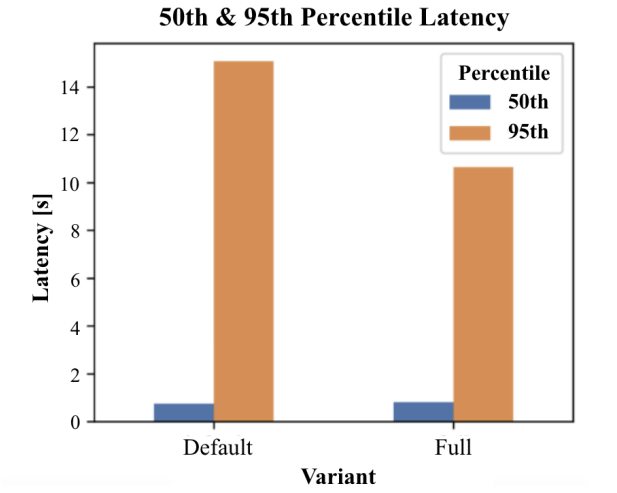
By including the default plan as an arm, the bandit inherently avoids catastrophic performance regressions. This is because if neither the IMC nor memory hints outperform the default, the default arm's posterior will dominate over time, causing the system to revert to the stock optimizer. Conversely, when the IMC or EIM arm yields consistent speedups, the bandit quickly learns to favor them. This not only balances exploration and exploitation more effectively than memory-only strategies, but also provides safeguards against performance regressions (offline flighting and heuristic filters cannot simultaneously achieve this without manual intervention).

3 PRELIMINARY EXPERIMENTS

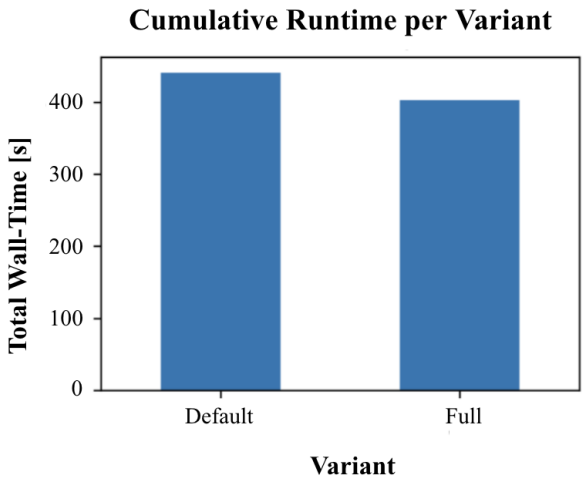
The following research questions are aimed to be addressed through the preliminary experiments:

(1) Can DataSwift truly prevent query regressions from occurring? If there are slowdowns, are any of them catastrophic towards overall latency?
(2) How well can DataSwift minimize the tail latency of the system? How well can it accelerate workloads?

To answer these questions, DataSwift was evaluated on PostgreSQL 16.1 [1] using the JOB Benchmark [8]along with queries from the



(a) Tail-latency distribution across percentiles.



(b) Total latency by variant.

Figure 2: Default refers to vanilla PostgreSQL timings while Full refers to DataSwift

Extended JOB Benchmark[1] [10] over the IMDb database [8]. Experiments were conducted on a virtual machine provisioned via Google Cloud, configured with 16 vCPUs, 96 GB of RAM, and a 200 GB SSD boot disk, running 64-bit Ubuntu 22.04 LTS. For baseline comparisons, vanilla PostgreSQL executes the JOB Benchmarks.

3.1 Performance Regressions

For the purposes of this investigation, a *performance regression* is quantified as any query whose execution under DataSwift is both at least 1.2× slower than vanilla PostgreSQL and adds at least 5 seconds in absolute latency [2]. Catastrophic slowdowns are those with at least a 3× slowdown and at least 50 seconds extra runtime. Out of 137 JOB queries, only one query (0.7% of the workload) meets the query slowdown criteria, and zero catastrophic slowdowns are observed. This demonstrates that DataSwift very rarely harms performance.

3.2 Latency Metrics

To understand performance across the workload, both median 50^{th} and tail 95^{th} percentile latencies are examined. *Tail latency* is defined as the latency at or above the 95^{th} percentile, or the slowest 5% of queries. Figure 2a plots the cumulative distribution across variants. At the median, DataSwift is ∼ 0.95x as fast as default PostgreSQL, but in the tail it achieves a **1.4x speedup** (41.5% faster) on the slowest queries. This slight increase in median latency reflects a tradeoff: the system defaults to conservative plans on simpler queries to avoid risking catastrophic slowdowns. By favoring robustness over marginal gains on already-fast queries, significant improvements in tail latency is achieved.

[1]The Extended JOB Benchmark was added to make the preliminary evaluation more rigorous

[2]Since almost all slowdowns experienced were only fractions of a second, this threshold helps filter out noise and focuses on cases where performance meaningfully degrades

Finally, aggregating end-to-end runtime over the JOB workloads (Fig. 2b), DataSwift yields a speedup of **∼1.1**x. These results show that while DataSwift may slightly increase latency on simple queries, it delivers significant improvements on the heaviest queries. This allows for meaningful reductions in total latency across the entire workload. Although some learned optimizers report faster speedups, such approaches heavily struggle with cold-start queries, retraining overhead, or unreliable uncertainty estimates (see Section 1). In contrast, DataSwift achieves end-to-end speedup without relying on repeated queries, offline replays, or high-latency retrieval mechanisms, making it better suited for deployments where safety is critical.

4 CONCLUSION AND FUTURE WORK

This paper has introduced DataSwift, an embedding-driven query optimization framework leveraging inductive matrix completion, learned embeddings from LLMs and GNNs, and Thompson-sampling bandit techniques. The preliminary results demonstrate that DataSwift substantially reduces tail latency, provides an overall speedup, and crucially avoids catastrophic performance regressions. Promising directions for future research include:

Repetitive Queries. Learned optimizers like LimeQO reduce total latency heavily through offline exploration on repetitive queries [15]. By precomputing optimal hint sets for repetitive queries while reserving DataSwift for novel queries, total latency can be significantly reduced.

Cost-Aware Adapter Fine Tuning. Small "adapter" modules can be inserted into each layer of the frozen Sentence-Transformer and trained on SQL-to-latency examples. These adapters can reshape the embeddings so that each coordinate reflects execution-cost factors. The resulting feature-set can speed up embedding extraction and improve accuracy. The exploration of these directions is left to future work—collaboration is welcome.

REFERENCES

[1] 2025. PostgreSQL Database. http://www.postgresql.org/. http://www.postgresql.org/

[2] Peter Akioyamen, Zixuan Yi, and Ryan Marcus. 2024. The Unreasonable Effectiveness of LLMs for Query Optimization. In *Proceedings of the Machine Learning for Systems Workshop at NeurIPS 2024 (Proceedings of Machine Learning and Systems)*, Vol. 6. 87–100. https://doi.org/10.48550/arXiv.2411.02862 arXiv:2411.02862 [cs.DB] To appear.

[3] Remmelt Ammerlaan, Gilbert Antonius, Marc Friedman, H. M. Sajjad Hossain, Alekh Jindal, Peter Orenberg, Hiren Patel, Shi Qiao, Vijay Ramani, Lucas Rosenblatt, Abhishek Roy, Irene Shaffer, Soundarajan Srinivasan, and Markus Weimer. 2021. PerfGuard: Deploying ML-for-Systems without Performance Regressions, Almost! *Proceedings of the VLDB Endowment* 14, 13 (2021), 3362–3375. https://doi.org/10.14778/3484224.3484233

[4] Christoph Anneser, Nesime Tatbul, David Cohen, Zhenggang Xu, Prithviraj Pandian, Nikolay Laptev, and Ryan Marcus. 2023. AutoSteer: Learned Query Optimization for Any SQL Database. *Proceedings of the VLDB Endowment* 16, 12 (2023), 3515–3527. https://doi.org/10.14778/3611540.3611544

[5] Baoming Chang, Amin Kamali, and Verena Kantere. 2024. A Novel Technique for Query Plan Representation Based on Graph Neural Nets. In *Proceedings of the 26th International Conference on Big Data Analytics and Knowledge Discovery (DaWaK 2024), Naples, Italy, August 26–28, 2024 (Lecture Notes in Computer Science)*, Vol. 14912. Springer, Cham, Switzerland, 299–314. https://doi.org/10.1007/978-3-031-68323-7_25

[6] Lyric Doshi, Vincent Zhuang, Gaurav Jain, Ryan Marcus, Haoyu Huang, Deniz Altinbüken, Eugene Brevdo, and Campbell Fraser. 2023. Kepler: Robust Learning for Faster Parametric Query Optimization. *Proceedings of the ACM on Management of Data* 1, 1, Article 109 (May 2023), 25 pages. https://doi.org/10.1145/3588963

[7] Amin Kamali, Verena Kantere, Calisto Zuzarte, and Vincent Corvinelli. 2024. Roq: Robust Query Optimization Based on a Risk-aware Learned Cost Model. arXiv preprint arXiv:2401.15210.

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

[9] 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)*. Association for Computing Machinery, New York, NY, USA, 1275–1288. https://doi.org/10.1145/3448016.3452838

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

[11] Nagarajan Natarajan and Inderjit S. Dhillon. 2014. Inductive matrix completion for predicting gene–disease associations. *Bioinformatics* 30, 12 (June 2014), i60–i68. https://doi.org/10.1093/bioinformatics/btu269 Open Access.

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

[13] 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 *Companion of the 2024 International Conference on Management of Data (SIGMOD/PODS '24)*. Association for Computing Machinery, New York, NY, USA, 280–294. https://doi.org/10.1145/3626246.3653391

[14] 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 (aiDM '24)*. Association for Computing Machinery, New York, NY, USA, 1–6. https://doi.org/10.1145/3663742.3663974

[15] Zixuan Yi, Yao Tian, Zachary G. Ives, and Ryan Marcus. 2025. Low Rank Learning for Offline Query Optimization. In *SIGMOD '25: 2025 ACM SIGMOD International Conference on Management of Data*. https://doi.org/10.48550/arXiv.2504.06399 arXiv:2504.06399 [cs.DB] To appear in SIGMOD 2025.

[16] 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 (2023), 1466–1479. https://doi.org/10.14778/3583140.3583160