Towards Industrial-Scale Parametric Query Optimization

Songsong Mo
Nanyang Technological University
Singapore
songsong.mo@ntu.edu.sg

Quanqing Xu*
OceanBase, Ant Group
Hangzhou, China
xuquanqing.xqq@oceanbase.com

Xuchen Ding
Nanyang Technological University
Singapore
xuchen002@e.ntu.edu.sg

Yue Zhao
Nanyang Technological University
Singapore
zhao0342@e.ntu.edu.sg

Zhifeng Bao
The University of Queensland
Queensland, Australia
zhifeng.bao@uq.edu.au

Chuanhui Yang
OceanBase, Ant Group
Hangzhou, China
rizhao.ych@oceanbase.com

Gao Cong*
Nanyang Technological University
Singapore
gaocong@ntu.edu.sg

ABSTRACT

Parametric Query Optimization (PQO) is crucial for efficiently executing parametrized queries (PQ) in modern industrial database systems. This paper addresses two key challenges overlooked by existing PQO techniques: large-scale template generalization and online adaptability. For large-scale workloads, maintaining one model per template is impractical, while sharing a single model across all templates leads to degraded performance. To overcome this issue, we propose a representation-based clustering strategy coupled with hierarchical model training, which significantly reduces model cost while preserving accuracy. For online adaptability, we observe that query parameter distributions shift over time, rendering fixed plan caches suboptimal. To address this, we introduce a KL-divergence-driven model fine-tuning and plan updating strategy that dynamically adapts to workload changes. Our approach is implemented on OceanBase and extensively evaluated on six workloads. Results show that it achieves up to 1.62× acceleration over the OceanBase optimizer and outperforms RankPQO, a state-of-the-art PQO method, by up to 1.23×, demonstrating improved scalability and robustness for industrial-scale PQO.

PVLDB Reference Format:
Songsong Mo, Quanqing Xu, Xuchen Ding, Yue Zhao, Zhifeng Bao, Chuanhui Yang, and Gao Cong. Towards Industrial-Scale Parametric Query Optimization. PVLDB, 19(12): 4303 - 4315, 2026.
doi:10.14778/3827998.3828034
PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/songsong945/RankPQO-industry.

1 INTRODUCTION

Parameterized Query Optimization (PQO) plays a crucial role in modern database systems, enabling efficient execution of queries with varying parameter bindings. Parameterized queries (PQ) are extensively employed in database applications, where the same SQL statement is executed repeatedly with different parameter values. A parameterized query, or *query template*, uses placeholders for constants, and the actual values supplied at runtime form a *parameter vector*. Many commercial database systems, such as Microsoft SQL Server [2] and OceanBase [1, 34], employ plan caching strategies to reduce optimization overhead by reusing precomputed execution plans. To further balance optimization efficiency and execution performance, a large body of work [6–8, 10–12, 15–17, 29, 31] has focused on the problem of Parametric Query Optimization.

In general, the PQO problem involves two key tasks: (1) selecting a small, diverse set of candidate plans to cache for a parameterized query offline, and (2) efficiently selecting the best plan from the cached set at runtime, based on the specific parameter values provided. Recent advancements in learning-based PQO methods [10, 29, 31] have demonstrated significant improvements by adopting a two-stage framework: candidate plan generation and best plan selection. In candidate plan generation, existing solutions either collect historical plans from query logs or generate new ones by adjusting cardinality estimates or join orders, retaining a subset of promising query plans as the candidate set. For best plan selection, they typically employ classification, regression, or learning-to-rank models to choose the most suitable plan from the candidate set for each incoming query instance. However, despite their effectiveness in controlled research environments, existing methods overlook key challenges in real-world industrial deployments.

One major challenge is scalability with a large number of query templates. Real-world database systems must support hundreds to thousands of templates per project (i.e., individual applications or business services in production systems), as observed in a production workload from OceanBase with over 5,000 distinct query templates (see Table 1). Existing methods provide two options that represent opposite extremes: training a separate model for each query template or using a single shared model across all templates. The former achieves good performance but incurs significant storage and training data collection overhead, making it impractical at scale. For example, training one model per template (RankPQO-NS [29]) requires more than 3.4 GB of storage for 3,300 templates (Table 2), and its data collection time grows linearly with the number of templates, exceeding 1,000 hours. The latter reduces storage

*Contact authors.

Proceedings of the VLDB Endowment, Vol. 19, No. 12 ISSN 2150-8097.
doi:10.14778/3827998.3828034

costs and training data collection time but suffers from degraded accuracy as the number of templates increases. As shown in Figure 2a, the single-model variant (RankPQO-S [29]) exhibits substantial performance degradation, with its speedup dropping from 2.34 at 33 templates to only 1.18 at 3,300 templates. Consequently, no existing method achieves both scalability and accuracy across a large set of templates.

Another challenge lies in adapting to evolving query parameter vector distributions, which shift over time due to changing user behavior, seasonal trends, or application updates. In this work, we focus on parameter distribution drift under the assumption of a relatively stable underlying data distribution. This assumption is consistent with prior workload evolution studies (e.g., Sibyl [14]), which evaluate time-evolving query patterns on fixed datasets without explicitly considering data distribution drift. Handling data distribution drift (e.g., changes in table statistics or data skew) is an important but complementary research problem and is left for future work. Existing methods typically train a static model and use it to select a subset of plans to cache, aiming to cover the current parameter distribution. However, as the parameter distribution drifts, two issues arise. First, although the relative quality of plans for the same parameter binding remains unchanged under a stable data distribution, the workload may increasingly contain parameter bindings or parameter combinations that were rarely observed during training. As a result, the learned ranking model may generalize poorly to these underrepresented parameter regions and choose suboptimal plans from the cached set. Second, the cached plan set itself may become suboptimal because it was selected to cover the original parameter distribution and may lack plans suited to the newly dominant parameter regions. Consequently, these approaches fail to maintain long-term performance under an evolving query parameter vector, with both RankPQO-S and RankPQO-NS suffering more than 30% performance degradation under distribution shifts (see Figure 2b). These two industrial challenges, scalability and adaptability, along with the limitations of existing solutions, motivate the design of a more robust PQO framework. These challenges and limitations are discussed in detail in Section 2.2 and Section 3.1.

To address the aforementioned challenges, we propose ScalePQO, a unified framework that integrates scalable template handling with adaptive online updating. Specifically, to tackle the scalability issue, we propose the idea of applying a clustering-based strategy that improves scalability without compromising prediction performance. For each cluster, we adopt the learning-to-rank formulation from RankPQO [29], which trains a model to compare candidate plans for a given query instance under different parameter vectors. The key novelty of our approach lies in reducing the number of required models through clustering, while preserving the effectiveness of ranking-based selection. A key challenge in this direction lies in how to effectively measure the similarity between query templates. As a baseline, we extend traditional edit-distance-based similarity into our framework. While edit-distance relies solely on query string structure, our clustering strategy leverages learned template embeddings to capture similarities that are correlated with query performance, thereby improving ranking accuracy within each cluster. Specifically, we first train a global rank model using all templates, then cluster templates based on the similarity of their learned embeddings, and finally fine-tune the global model on the data of each cluster to obtain specialized models. This process enables model sharing across similar templates and provides a balance between accuracy and efficiency.

To address parameter distribution drift, one straightforward solution is directly retraining the model and re-selecting candidate plans at every time slot. Here, a time slot denotes a logical batch of query instances grouped by arrival orders to monitor distribution drift, and queries inside each slot are still executed and responded to in real time. However, this brute-force solution is impractical even if performed in the background, since full retraining and plan re-selection introduce prohibitive latency and resource consumption in online settings. To this end, we propose a KL-divergence-based online adaptation mechanism. Specifically, ScalePQO computes the KL divergence between the two most recent slots, which adaptively guides the extent of model fine-tuning and cached plan updating performed in the background while the current slot is being served. We refer to these two procedures as adaptive model fine-tuning and cached plan updating. The updated model and plan set are then deployed at the beginning of the following slot. This design enables low-overhead and timely adaptation, allowing ScalePQO to maintain robust performance under evolving workloads in real-world systems such as OceanBase.

In summary, our contributions are as follows:

- We analyze the key challenges of industrial-scale PQO and propose a novel framework, ScalePQO, to address them, bridging the gap between research-driven PQO methods and real-world database deployments.
- We introduce a representation-based template clustering strategy that enables cluster-level model sharing for PQO. Unlike prior work that focuses on clustering within a single template's parameter–plan space, our method clusters across templates to improve scalability while maintaining competitive prediction accuracy. To handle evolving query parameter vector distributions, we further propose a KL-divergence-based online adaptation mechanism, which includes adaptive model fine-tuning and cached plan updating.
- We construct benchmark workloads whose parameter distributions follow those observed in real production projects in OceanBase. Extensive experiments on OceanBase demonstrate that our approach significantly improves both optimization efficiency and query execution performance compared to existing PQO solutions.

2 CHALLENGES FOR INDUSTRIAL DEPLOYMENT OF RANKPQO

In this section, we analyze the challenges encountered when deploying our previously proposed approach, RankPQO [29], in OceanBase. Before that, we first provide a brief introduction to the parametric query optimization (PQO) problem, followed by an overview of the default solution of OceanBase and RankPQO.

2.1 PQO in OceanBase and RankPQO

Parametric query optimization problem. Parametric Query Optimization (PQO) aims to optimize the execution of parameterized queries, where the same query template runs with different

parameter values [10, 29, 31]. A parameterized query typically uses placeholders in the `WHERE` clause, and at runtime, users supply specific values as a parameter vector. PQO addresses the challenge of selecting efficient execution plans across a wide range of possible parameter bindings.

PQO consists of two key sub-problems:

- **Candidate Plan Generation:** This task involves generating and selecting a small set of representative query execution plans for each query template. These plans are cached offline to avoid repeated optimization at runtime. Since the plan space can be very large (due to different join orders, access paths, and physical operators), the selected plans should collectively cover diverse query behaviors across the parameter space, while keeping the total cache size under control.
- **Best Plan Selection:** Given an incoming query instance with specific parameter values, this step selects the most efficient plan from the cached candidates. The goal is to minimize execution latency by matching the parameter vector to the most suitable plan. This step is challenging because the relationship between parameter values and plan performance is often complex and non-linear.

PQO in OceanBase. OceanBase [34] adopts a plan caching mechanism to optimize parameterized queries. When a query instance arrives, the system attempts to match its parameter selectivity with an existing plan in the cached plan set. If no matching plan is found (cache miss), a new execution plan is generated and added to the plan cache, corresponding to Candidate Plan Generation. If a suitable plan exists (cache hit), it is reused to avoid re-optimization overhead, corresponding to Best Plan Selection. However, a key limitation of this approach is that the number of cached plans continuously increases as more queries are executed, leading to an unbounded cache growth without a mechanism to refine or remove suboptimal plans. This can result in inefficient plan selection and increased memory consumption.

RankPQO. RankPQO [29] is a learning-based approach to parametric query optimization (PQO) that improves both candidate plan generation and best plan selection. Unlike traditional methods that rely on heuristic-based cost estimation, RankPQO formulates plan selection as a learning-to-rank problem. It consists of four key components: plan enumeration, model training, candidate plan selection, and best plan selection.

Plan Enumeration. RankPQO adopts a hybrid enumeration strategy that efficiently generates a diverse set of candidate plans by jointly varying cardinality estimates and join orders. Prior results have shown that this approach can produce a large number of distinct plans within a limited time budget, enhancing the coverage of different parameter bindings.

Model Training. To capture the query-dependent performance of execution plans, RankPQO trains a pairwise ranking model using execution latency collected from sample workloads. This latency serves as the sole supervision signal. Operator-level statistics (e.g., operator types, tables accessed, and predicates), along with other plan-structure features, are used only as input features for plan encoding. The model consists of three main components: parameter_net, plan_net, and comparator. Specifically, parameter_net encodes input parameter vectors into dense representations, while plan_net converts the structure of each query plan into a plan embedding, typically using a tree-based encoder. The comparator then computes the similarity between each plan embedding and the parameter embedding to predict relative performance. During training, given a parameter vector $\mathbf{v}$ and a pair of plans p_1, p_2 with observed latency labels, the model is optimized to predict the correct preference order. By learning to rank plans relative to each parameter context, the model avoids reliance on inaccurate cost models and generalizes better across varying workloads.

Candidate Plan Selection. Using the trained model, RankPQO selects a set of k candidate plans to cache. A greedy selection algorithm is applied to identify plans that offer high coverage and low latency across query instances. This process relies on the model's ranking scores to guide plan selection under the caching constraint.

Best Plan Selection. At runtime, RankPQO uses the trained ranking model to score all cached plans for a given query instance and selects the top-ranked plan. The selected plan is translated into a hint using `pg_hint_plan` [4] to guide the database engine toward executing the desired plan.

While RankPQO has demonstrated effectiveness in controlled environments, deploying it in OceanBase presents additional challenges, including handling a large scale of query templates and adapting to evolving query parameter distributions, which we address in the following sections.

Table 1: Query template distribution across projects

Project	P1	P2	P3	P4	P5	P6
Templates	5047	2342	895	1578	1363	1835

2.2 Scalability and Adaptability Challenges

Recent learning-based PQO methods [10, 29, 31] have shown promising results in controlled environments. However, they have typically been evaluated on workloads with fewer than 100 query templates per workload and assume a static parameter distribution. As a result, directly applying these methods in industrial settings introduces additional scalability and adaptability challenges.

To illustrate these challenges, we analyze a real-world workload from OceanBase. Table 1 presents the distribution of query templates across multiple projects (i.e., individual applications or business services in production systems), showing that each project contains hundreds to thousands of unique templates. Given this scale, maintaining a separate model for each template is impractical due to excessive storage and computational overhead. Moreover, Figure 1 tracks the KL divergence of query parameter distributions for a representative template over time, revealing significant distribution shifts. The detailed definition and computation of KL divergence are provided in Section 5. This shift mainly stems from two sources: user variation and query content evolution. For example, different users may exhibit different parameter distributions, and even the same user's query behavior can change over time. For instance, a user might search for clothing today and electronics tomorrow.

These observations highlight two key industrial challenges. Challenge 1: Scalability with Large Query Templates. Industrial workloads consist of a vast number of query templates, making it impractical to train a separate model for each template due to excessive

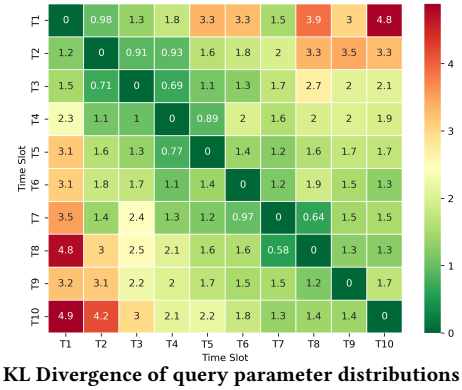


Figure 1: KL Divergence of query parameter distributions over time

storage and computational costs. However, using a single model for all templates results in degraded performance as the number of templates increases. A scalable approach is needed to efficiently cluster templates and share models while maintaining performance. Challenge 2: Adapting to Parameter Distribution Shifts. Query parameter distributions evolve due to user behavior changes, seasonal trends, and application updates. As a result, cached plans gradually become suboptimal, leading to performance degradation. To address this, an adaptive mechanism is required to fine-tune models and update cached plans dynamically in response to workload shifts.

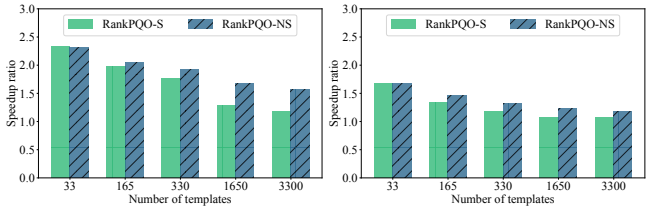
While RankPQO and similar learning-based PQO methods improve upon traditional plan caching strategies, they still face critical limitations in industrial deployments. For Challenge 1, training a separate model for each template leads to high storage overhead. However, using a single model for all templates results in performance degradation as the number of templates increases. Thus, a key challenge is balancing model storage costs and end-to-end performance. For Challenge 2, the plan caching mechanism of OceanBase does not naturally converge to a fixed set of cached plans, leading to uncontrolled cache growth. For learning-based PQO methods such as RankPQO, a fixed cached plan set becomes less effective over time due to shifts in parameter distributions, progressively degrading performance. This motivates our framework, which introduces a scalable template clustering mechanism and an adaptive fine-tuning strategy to efficiently handle large-scale query templates and evolving workloads.

3 THE SCALEPQO FRAMEWORK

To address the industrial deployment challenges discussed in Section 2.2, we propose the ScalePQO framework. It is designed to improve scalability with large template sets and adapt to evolving parameter vector distributions. We begin by studying the scalability and adaptability limitations of directly deploying RankPQO in industrial settings, which motivates the key components of ScalePQO.

3.1 Scalability and Adaptability Study of RankPQO

To better understand the limitations of RankPQO in industrial scenarios, we conduct a comprehensive evaluation on two key aspects: scalability with large template sets and adaptability under evolving parameter distributions.



(a) Scalability Test **(b) Adaptability Test**

Figure 2: Scalability and adaptability study of RankPQO

Training data collection strategy. Before model training, we need to collect training data consisting of parameter vector and their corresponding plan labels. Following the original RankPQO setting, a straightforward strategy is to collect a fixed number of training samples per query template (e.g., 3000). However, this approach causes the total data collection time to grow linearly with the number of templates.

In our experiments, we vary the number of templates from 33 to 3300. Collecting data for 33 templates takes approximately 13 hours. Therefore, collecting data for 3300 templates would require about 1300 hours, which is clearly impractical for industrial deployment. To address this, we fix the total number of training samples by maintaining a constant product of the number of templates and the number of training samples per template. This ensures the overall data collection time remains stable (around 13 hours per dataset) regardless of the number of templates.

Scalability analysis. We conduct a scalability analysis by varying the number of query templates while keeping a fixed set of static parameter vectors. For each configuration, we split the parameter vectors into 80% for training and 20% for testing. Figure 2a shows the end-to-end performance of RankPQO across different template scales. The speedup ratios reported in Figure 2 are all computed relative to the original OceanBase optimizer. We compare two variants: RankPQO-S, which uses a single shared model for all templates, and RankPQO-NS, which trains one model per template.

As the number of templates increases, RankPQO-NS exhibits a slight decline in performance, primarily due to fewer training samples per template under a fixed total training budget. Moreover, this per-template modeling approach incurs substantial storage overhead, as shown in Table 2. For example, at 3300 templates, the total model size reaches over 3.4 GB. In contrast, RankPQO-S, which uses a single shared model, suffers from a more significant performance drop. This suggests that a single model struggles to generalize across a large number of diverse templates, especially when they vary significantly in structure and parameter semantics.

Adaptability analysis. We further evaluate the adaptability of RankPQO under dynamic workloads, where parameter distributions shift over time. In real-world database systems, user behavior or upstream application logic may evolve, causing query parameter patterns to change gradually or abruptly. To simulate such scenarios, we design synthetic drifts by shifting the parameter sampling range across consecutive time slots.

Figure 2b presents the performance of RankPQO-S and RankPQO-NS under parameter distribution shift, while varying the number of query templates. Both methods exhibit noticeable degradation, confirming that fixed models trained on static data are not robust to distribution drift. RankPQO-NS shows a milder degradation trend due

Table 2: Model size (MB) under different numbers of templates

	33	165	330	1650	3300
RankPQO-S	5.9	25.9	51	251.4	502
RankPQO-NS	34.3	171.6	343.2	1716	3432

to its per-template specialization, which provides some resilience to shifts within a specific template's parameter space. However, it still suffers as the training data becomes stale. RankPQO-S, on the other hand, struggles to generalize across templates and across time. Since it encodes a single shared mapping from parameters to plans, it is particularly sensitive to distribution mismatch between training and testing phases. These results highlight the need for adaptability mechanisms such as periodic model fine-tuning and selective plan updates, which we incorporate in ScalePQO.

Model size comparison. In addition to execution performance, we compare the memory footprint of RankPQO variants. Table 2 reports the total model size (in MB) under different template counts. RankPQO-NS exhibits linear growth in total size, as each new template adds a dedicated model. For example, at 3300 templates, the total model footprint exceeds 3.4 GB, posing a significant burden for deployment and storage management.

In contrast, RankPQO-S maintains a constant model size regardless of template count, since it uses a single shared model. This results in a much smaller footprint, about 502 MB at most, which is preferable for memory-constrained environments. However, this compactness comes at the cost of reduced accuracy, as shown in Figure 2a. Thus, while RankPQO-NS is more accurate, it fails to scale in storage cost; RankPQO-S is scalable in size but lacks robustness and generalization ability. This trade-off between scalability and performance motivates the need for a new solution that can support large-scale PQO with controlled model complexity and improved adaptability. Our proposed method addresses this by combining cluster-aware modeling and dynamic updates.

3.2 Overview of ScalePQO

To address the challenges discussed earlier, we propose ScalePQO, a two-stage framework that integrates offline training with online deployment, as illustrated in Figure 3. Compared to RankPQO, our framework introduces two new components (highlighted in green): a template clustering module (Section 4) to improve scalability, and a KL-divergence-driven online adaptation mechanism (Section 5) to improve robustness under evolving workloads.

In the offline stage, we first cluster query templates based on structural and execution similarity. Instead of training a separate model for each template, we train a shared ranking model for each cluster, which significantly reduces storage and data collection costs while preserving performance. For each cluster, we generate candidate plans and collect training data for rank model learning.

In the online stage, we partition query execution into discrete time slots. During each slot, the system uses the deployed models and cached candidate plans for real-time best plan selection. For an incoming query instance, the system first identifies its corresponding template and extracts the parameter vector. The ranking model embeds both the parameter vector and the cached candidate plans of that template (or cluster), scores them, and selects the top-ranked plan. Each cached plan encodes a specific join order and operator configuration. The selected plan is then translated into optimizer hints to guide the traditional query optimizer to generate the final executable physical plan for execution. Concurrently, in the background, the system monitors parameter distribution shifts across slots. When a significant change is detected (measured via KL divergence), we trigger model fine-tuning and plan updating. Both steps are executed concurrently with query processing, providing low overhead and timely adaptation. The updated model and plan set are used in the subsequent time slot.

By combining scalable clustering and adaptive model updating, ScalePQO effectively balances model size, training overhead (i.e., training data collection costs), and query performance, making it suitable for industrial-scale deployment in systems such as OceanBase.

4 OFFLINE TEMPLATE CLUSTERING AND MODEL TRAINING

In this section, we present a template clustering mechanism and describe how we train a rank model for each cluster. SQL query clustering [19] primarily focuses on defining similarity or distance between SQL queries, followed by the application of existing clustering algorithms such as k-means. Traditional approaches typically rely on query features, such as selection predicates and parameter selectivity, join conditions, and group-by clauses, to compute similarity. However, unlike SQL queries, query templates do not have bound parameters, making it difficult to directly capture complex query features, such as selection predicates and parameter selectivity. To address this, we introduce an edit-distance-based method as a baseline for measuring template similarity. Since our goal is to improve the effectiveness of rank models within each cluster, we further propose a similarity metric based on rank model embeddings, which captures deeper query relationships beyond structural similarity. Finally, we integrate this clustering process with model training and introduce a hierarchical training approach, which enhances efficiency and scalability.

We use k-means as the clustering algorithm in both clustering strategies. The purpose is not to claim that k-means is theoretically optimal for these spaces, but to use it as a practical and scalable partitioning tool for cluster-level model specialization. K-means tends to produce relatively balanced clusters, which helps maintain sufficient training data for each cluster-level rank model. In the edit-distance-based baseline, k-means is applied over pairwise structural distances to group syntactically similar templates. In the embedding-based approach, it is applied to learned template representations that capture performance-relevant similarities. Although these spaces may not strictly satisfy the isotropic cluster assumption of k-means, the results in Table 5 show that this design is effective in practice.

4.1 Edit-distance-based Baseline

The framework of the edit-distance-based solution is straightforward. It first computes the edit distance between query templates and then applies k-means clustering to partition them into balanced groups. The use of k-means ensures that clusters remain relatively balanced, preventing an excessive number of small or unbalanced clusters. This design choice helps maintain the performance of the rank model, as it avoids degradation due to an overwhelming

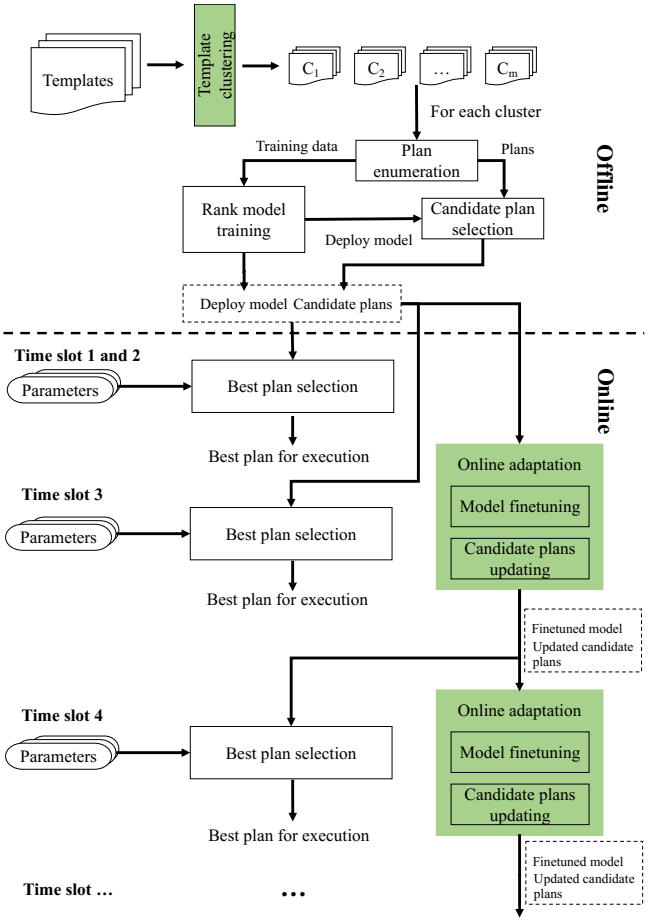


Figure 3: Framework

number of query templates in a single cluster. After clustering, we follow the same process as RankPQO [29] to train one rank model per cluster, which we omit here for brevity.

We compute edit distance at the clause level rather than directly over the entire SQL string. Direct string-level edit distance can be sensitive to superficial formatting differences, clause ordering, aliases, and syntactic variations that may not reflect structural similarity. By decomposing a query template into major SQL clauses, this baseline captures structural differences in a more interpretable way while remaining simple and lightweight. We do not intend this baseline to be an optimal SQL similarity measure. Instead, it serves as a simple structural baseline for comparison with our embedding-based clustering method. Our current implementation focuses on SPJ-style query templates used in the evaluated PQO workloads. Supporting more complex SQL structures, such as nested subqueries or UNION ALL, would require recursively comparing subquery blocks or set-operation branches, which we leave for future work.

Algorithm 1 presents the pseudo-code for edit-distance-based query template clustering. Initially, we compute the pairwise edit distance matrix D to quantify the structural differences between query templates (lines 1-5). The function calculate_edit_distance decomposes each query template into five key components: S (SELECT clause), F (FROM clause), W (WHERE clause), G (GROUP BY clause), and O (ORDER BY clause), and then computes the sum of

Algorithm 1: Edit-Distance-Based Clustering($\mathcal{T}$, c)

Input: $\mathcal{T}$: set of query templates, c: number of clusters
Output: C: set of clusters

1 Initialize distance matrix D of size $|\mathcal{T}| \times |\mathcal{T}|$;
2 **for** $i = 1$ *to* $|\mathcal{T}|$ **do**
3 **for** $j = i + 1$ *to* $|\mathcal{T}|$ **do**
4 $D[i, j] \leftarrow$ calculate_edit_distance($\mathcal{T}_i, \mathcal{T}_j$);
5 $D[j, i] \leftarrow D[i, j]$;
6 ClusterLabels $\leftarrow$ KMeans(D, c);
7 Initialize empty set $C \leftarrow \{C_1, C_2, \ldots, C_c\}$;
8 **for** $i = 1$ *to* $|\mathcal{T}|$ **do**
9 $C[\text{ClusterLabels}[i]] \leftarrow C[\text{ClusterLabels}[i]] \cup \{\mathcal{T}_i\}$;
10 **Return** C;
11
Function calculate_edit_distance($\mathcal{T}_i, \mathcal{T}_j$):
12
13 Parse $\mathcal{T}_i$ and $\mathcal{T}_j$ into components: SelectClause S, FromClause F, WhereClause W, GroupByClause G, OrderByClause O;
14 Initialize edit distance $E \leftarrow 0$;
15 **if** $\mathcal{T}_i$ *and* $\mathcal{T}_j$ *have different DISTINCT usage* **then**
16 $E \leftarrow E + 1$;
17 $E \leftarrow E + d_{\text{lev}}(S_i, S_j) + d_{\text{lev}}(F_i, F_j) + d_{\text{lev}}(W_i, W_j)$
18 $+ d_{\text{lev}}(G_i, G_j) + d_{\text{lev}}(O_i, O_j)$;
19 **return** E;

the Levenshtein distances [22] for each component. Subsequently, we apply k-means clustering on D to partition the templates into c clusters (line 6). Finally, each template is assigned to the nearest cluster to ensure that structurally similar queries are grouped together (lines 7-9). The resulting clusters $C_1, C_2, \ldots, C_c$ serve as the final output.

4.2 Hierarchical Model Training via Template Embedding Clustering

Recalling Section 3.2, our primary motivation for clustering is to enhance the performance of the rank model within each cluster, rather than merely grouping structurally similar templates together, as the baseline method (Section 4.1) does. The baseline approach relies on edit-distance-based similarity, which captures syntactic closeness but does not necessarily reflect the optimization characteristics learned by the model. Therefore, we propose a model-embedding-based clustering approach, where query templates are grouped based on their learned representations in the model space. The intuition behind this approach is that if multiple templates are mapped to the same embedding region, it implies that the model lacks sufficient discrimination ability within that region. To address this issue, we zoom in by fine-tuning separate models for each cluster, allowing for more precise ranking and plan selection.

Importantly, our clustering is not driven by SQL textual similarity. While edit distance serves as a structural baseline in Section 4.1, it may group templates that are syntactically similar but exhibit different execution behaviors under varying parameter bindings. In contrast, our method clusters templates based on their learned embeddings derived from the ranking model, which capture execution-related characteristics and parameter sensitivity. Therefore, the

clustering signal reflects optimization behavior rather than purely syntactic structure. To seamlessly integrate clustering into the training workflow, we adopt a hierarchical training process. We first train a single model for all query templates, enabling it to learn high-level query template representations. Then, we use the model to compute template embeddings, which serve as feature representations in the learned model space. Next, we apply k-means clustering to partition templates based on their embeddings, grouping those with similar learned representations. Finally, we fine-tune a separate model for each cluster using its corresponding query execution data, allowing the model to better adapt to workload variations within each cluster.

The detailed procedure is outlined in Algorithm 2. The process begins by training a global model using execution data collected from all query templates (lines 1-6). Each query template is paired with a set of parameter values and execution plans, and their corresponding query latencies are recorded to construct the training dataset. Next, we compute template embeddings based on the trained model (line 7). These embeddings encode both parameter and execution plan information, allowing us to measure the similarity between templates in a learned representation space. We use mean pooling to obtain a lightweight template-level representation for scalable clustering. This representation summarizes the overall parameter and plan behavior of each template, while keeping the clustering process efficient for large-scale workloads. The goal of this step is to obtain a coarse-grained partitioning for cluster-level model specialization, rather than to fully model all intra-template parameter modes. Using these embeddings, we perform k-means clustering to partition query templates into c clusters (lines 8-11). This approach groups templates with similar learned representations, facilitating more effective specialization in later stages. Finally, we fine-tune a separate model for each cluster using its respective execution data (lines 12-15). This step enables the model to adapt to workload variations within each cluster, improving ranking effectiveness. The final output consists of a set of fine-tuned models and the corresponding template clusters, which together form the foundation for optimizing parametric queries.

Algorithm 2: Hierarchical Model Training via Template Embedding Clustering

Input: $\mathcal{T}$: set of query templates, c: number of clusters
Output: C: set of clusters, $\mathcal{M}$: set of fine-tuned models

1 Initialize training dataset $\mathcal{D} \leftarrow \emptyset$;
2 **foreach** $\mathcal{T}_i \in \mathcal{T}$ **do**
3 Generate parameter values $\mathcal{V}_i$ and execution plans $\mathcal{P}_i$;
4 Collect execution latency $\mathcal{L}_i$ for $(\mathcal{V}_i, \mathcal{P}_i)$;
5 $\mathcal{D}_i \leftarrow \{(\mathcal{V}_i, \mathcal{P}_i, \mathcal{L}_i)\}$; $\mathcal{D} \leftarrow \mathcal{D} \cup \mathcal{D}_i$;
6 Train global model M using $\mathcal{D}$;
7 Template embeddings $E = \{E_i \leftarrow \text{compute\_embedding}(M, \mathcal{V}_i, \mathcal{P}_i) \mid \mathcal{T}_i \in \mathcal{T}\}$;
8 ClusterLabels $\leftarrow$ KMeans(E, c);
9 Initialize clusters $C = \{C_1, \ldots, C_c\}$;
10 **foreach** $\mathcal{T}_i \in \mathcal{T}$ **do**
11 Assign $\mathcal{T}_i$ to cluster $C[\text{ClusterLabels}[i]]$;
12 Initialize fine-tuned models $\mathcal{M} \leftarrow \emptyset$;
13 **foreach** $C_j \in C$ **do**
14 Extract training data $\mathcal{D}_j = \{\mathcal{D}_i \mid \mathcal{T}_i \in C_j\}$;
15 Fine-tune model M_j using $\mathcal{D}_j$; $\mathcal{M} \leftarrow \mathcal{M} \cup M_j$;
16 **Return** C, $\mathcal{M}$;
17 ---
Function `compute_embedding`$(M, \mathcal{V}, \mathcal{P})$**:**
18 ---
19 Convert $\mathcal{V}$ into tensor format;
20 ParamEmbs $\leftarrow$ M.parameter_net$(\mathcal{V})$;
21 AvgParamEmb $\leftarrow$ mean$(ParamEmbs)$;
22 PlanTrees $\leftarrow$ M.build_trees$(\mathcal{P})$;
23 PlanEmbs $\leftarrow$ M.plan_net$(PlanTrees)$;
24 AvgPlanEmb $\leftarrow$ mean$(PlanEmbs)$;
25 TemplateEmb $\leftarrow$ concat$(AvgParamEmb, AvgPlanEmb)$;
26 **return** TemplateEmb;

5 ONLINE MODEL FINE-TUNING AND QUERY OPTIMIZATION

During the query execution (the online stage), RankPQO [29] selects for each query instance the best execution plan from the cached query plans using a learned ranking model. However, as discussed in Section 3.2, real-world deployments demonstrate that query parameter distributions evolve over time, leading to distribution shifts that degrade both model prediction accuracy and the effectiveness of cached query plans.

To address this challenge, we partition the query timeline into time slots and perform periodic model fine-tuning and candidate plan updating within each slot to adapt to parameter distribution shifts. Since this process occurs during online deployment, the key issue is how to efficiently update the model and candidate plans without incurring excessive computational overhead. To this end, we propose an adaptive update strategy driven by KL-divergence, which dynamically adjusts both model fine-tuning and candidate plan updating in response to workload shifts. Crucially, our approach executes model adaptation and plan updates concurrently with query processing, ensuring uninterrupted online service.

In Section 5.1, we present our method of quantifying parameter distribution shifts based on KL divergence. Based on this, Section 5.2 introduces an efficient strategy for model fine-tuning, and Section 5.3 presents our approach for candidate plan updating.

5.1 Parameter Shift Detection

To quantify the evolution of query parameter distributions within each cluster, we propose to employ the Kullback-Leibler (KL) divergence to compute the difference between distributions over consecutive time slots. Given a cluster C consisting of n query templates, where each template $\mathcal{T}_i$ has m parameters, we compute the KL divergence for each parameter and aggregate the results.

Probability estimation for different parameter types. Since query parameters have different data types, we adopt different strategies to estimate their probability distributions. For numerical and temporal parameters, we apply histogram binning. Specifically, the value range is divided into fixed-width bins, and the normalized counts in each bin are treated as a discrete probability distribution. For categorical parameters, we compute the empirical distribution

by counting the occurrences of each unique value and normalizing the frequencies.

KL divergence computation. Let $\mathcal{V}_{T_i}$ and $\mathcal{V}_{T_j}$ denote the set of user-input parameter vectors within time slots T_i and T_j, respectively. Since each query template has a fixed set of m parameters, we compute the KL divergence for each parameter dimension k:

$$D_{\mathrm{KL}}^{k}(P_k \parallel Q_k) = \sum_{x} P_k(x) \log \frac{P_k(x)}{Q_k(x)}, \tag{1}$$

where $P_k(x)$ and $Q_k(x)$ represent the probability distributions of parameter k in time slots T_i and T_j, respectively. To obtain an overall measure for a cluster, we aggregate the KL divergence across all parameter dimensions by computing:

$$D_{\mathrm{KL}}(\mathcal{V}_{T_i} \parallel \mathcal{V}_{T_j}) = \frac{1}{m} \sum_{k=1}^{m} D_{\mathrm{KL}}^{k}(P_k \parallel Q_k), \tag{2}$$

where m is the number of parameters in the parameter vector.

5.2 Model Fine-Tuning

To overcome these limitations, we propose a KL-divergence-based online adaptation mechanism that selectively triggers model updates only when significant distribution shifts are detected. This adaptive strategy reduces overhead while maintaining robustness, making it more suitable for dynamic, large-scale production environments. Specifically, we apply two strategies: (1) We freeze the plan embedding layers and update only the parameter embedding and ranking layers, as only parameter distributions evolve over time; (2) We dynamically adjust the number of fine-tuning steps based on the magnitude of KL divergence between time windows, allowing efficient and workload-aware adaptation. To avoid additional serving latency, fine-tuning is executed concurrently with query processing during time slot T. The updated model is then deployed at the beginning of time slot T+1.

Adaptive fine-tuning steps. To efficiently balance model adaptation and computational cost, we dynamically adjust the number of fine-tuning steps based on the KL divergence. Given the computed divergence D_{KL} between the parameter distributions in consecutive time slots, we determine the number of training iterations as follows:

$$\text{steps} = \left\lceil N_{\max} \times \min\left(1, \frac{D_{\mathrm{KL}}}{\tau_{\max}}\right) \right\rceil, \tag{3}$$

where $N_{\max}$ is the maximum number of fine-tuning steps, and $\tau_{\max}$ is the upper threshold for full fine-tuning. If $D_{\mathrm{KL}} \geq \tau_{\max}$, the model undergoes a complete fine-tuning cycle with $N_{\max}$ steps. Otherwise, the training steps are scaled proportionally to the divergence ratio.

Fine-tuning workflow. To reduce computational overhead, we selectively update only the parameter embedding layers (θ_{param}) and ranking layers (θ_{rank}), while keeping plan embedding layers (θ_{plan}) unchanged. KL divergence is computed based on the parameter distributions from time slots $T-2$ and $T-1$. The number of fine-tuning steps is then determined accordingly and applied using execution data collected via *what-if* calls during time slot T. This fine-tuning process runs in parallel with query serving, so that the updated model becomes available at the beginning of time slot T+1 without introducing delays. This aligns with the goal of enabling adaptive updates without interrupting query processing. The process starts from $T = 3$ to provide sufficient historical information for divergence estimation.

5.3 Candidate Plan Updating

As query parameter distributions evolve, previously cached plans may become suboptimal. To maintain query performance, we introduce an adaptive plan updating strategy guided by KL divergence. Instead of replacing the entire plan set, we dynamically adjust the number of plans to be updated based on the magnitude of the distribution shift.

Adaptive number of plan updates. Following model fine-tuning, we perform plan updating within the same time slot T. KL divergence D_{KL} is computed using the parameter distributions from time slots $T-2$ and $T-1$, capturing recent workload shifts. Based on this, the number of plans to update, denoted as r, is determined as:

$$r = \left\lceil k \times \min\left(1, \frac{D_{\mathrm{KL}}}{\tau_{\max}}\right) \right\rceil, \tag{4}$$

where k is the total number of cached plans, and $\tau_{\max}$ is the threshold for full plan updating. This updating is executed concurrently with query processing, without blocking online serving. The updated plan set becomes effective at the beginning of slot T+1.

Plan updating strategy. Using the fine-tuned model from time slot T, we generate a new set of candidate plans and select the top-r plans from this set. These new plans replace the bottom r entries in the current cache, while the top-$(k-r)$ plans from the existing cache are retained. The updated cache is reordered such that new plans are inserted at the front to prioritize recently optimized selections. As with model fine-tuning, plan updating is performed concurrently with query serving during slot T, and the updated plan cache becomes active at the beginning of time slot T+1.

6 EXPERIMENTS

In this section, we evaluate the performance of ScalePQO in comparison to baselines regarding execution latency speedups on parameterized query workloads.

6.1 Experimental Setup

Datasets and Workloads. Due to user privacy agreements, we cannot publish or directly expose internal OceanBase workloads. To provide a more comprehensive evaluation, we use three types of workloads: one anonymized internal OceanBase workload, four synthetic workloads calibrated using production observations, and one public StackExchange-based workload.

Anonymized workload. We use an anonymized workload, denoted as Anonymized, collected from the OceanBase production project P3, which contains 895 query templates. For each template, we collect 50 parameter bindings and evaluate them on a 1GB anonymized database snapshot. All sensitive information, including table names, column names, business semantics, and concrete data values, is anonymized before evaluation.

Synthetic workloads. We construct four synthetic query workloads that reflect the scale and complexity observed in real-world industrial settings. In particular, these synthetic workloads are

designed to follow the parameter distributions observed in real production projects in OceanBase, including realistic parameter drift patterns over time. We conduct experiments on two benchmark datasets: the Internet Movie Database (IMDB) [21] and the DSB benchmark [9]. For DSB, we set the scale factor to 10. Based on each dataset, we construct two synthetic workloads from JOB [21] and DSB [9] query workloads, resulting in four workloads: JOB330, JOB3300, DSB150, and DSB1500. The detailed generation process for these synthetic query templates and parameters is described in [28].

Stack workload. To further evaluate ScalePQO on a public real-world workload, we construct a StackExchange-based workload, Stack for short, using publicly available StackExchange data [3]. We collect 150 non-duplicate popular queries from StackExchange Data Explorer as query templates. For each template, we collect 50 parameter bindings from recent queries with the same normalized template structure, where constants in predicates are replaced with parameter placeholders. We evaluate this workload on a 3GB StackExchange database. Unlike the synthetic workloads, the query templates and parameter bindings in this workload are derived from real user-written StackExchange queries.

Baselines. We compare ScalePQO against the production-grade database system OceanBase and several learning-based query optimizers for PQO. Our main baseline is RankPQO [29], the state-of-the-art framework for parametric query optimization. Specifically, we evaluate two RankPQO variants: *RankPQO-S*, which uses a single shared model for all query templates, and *RankPQO-NS*, which trains an individual model for each template. These two variants highlight the trade-off between scalability and performance discussed in Section 3.1. We also include *Kepler* [10], a learning-based optimizer that heuristically adjusts sub-plan cardinalities to generate diverse plans. Prior work [29] has shown that RankPQO outperforms LogPQO [31], so we omit LogPQO from our experiments. To compare against non-learning approaches, we include *Density* [6], a clustering-based parametric plan caching method that predicts plans based on local density estimation in the optimizer parameter space. We select Density as it represents clustering-based PQO techniques, enabling a direct comparison with clustering-based strategies. Density models the optimizer parameter space and predicts plans using local neighborhood density estimation with a confidence threshold. Finally, we include *OptOnce*, a simple plan caching strategy that stores the first generated plan for each query template and reuses it for all subsequent parameter instances without re-optimization.

Metrics and Default Parameter Settings. To evaluate performance, we adopt the speedup ratio as the key metric, defined as the runtime of OceanBase divided by the runtime of the evaluated method. Unless otherwise specified, we use the following default settings throughout the experiments. Following the configuration in RankPQO [29], we set the number of cached candidate plans to 30, train models using the Adam optimizer with a learning rate of 0.001, and set the number of training epochs to 10. For training data collection, we also adopt a timeout-based strategy using 12 CPU cores, which requires approximately 13 hours per dataset. Additionally, we use default parameter settings in our experiments. The number of template clusters is set to $c = 10$ for JOB330 and DSB150, and $c = 50$ for JOB3300 and DSB1500. The KL divergence threshold for triggering online adaptation is set to $\tau_{\max} = 2$. For Density, we set the neighborhood radius to 0.1 and the confidence threshold to 0.5.

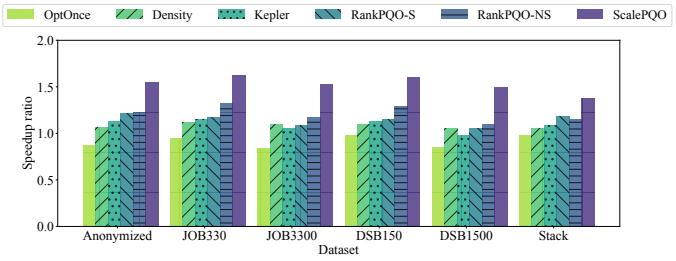


Figure 4: Varying dataset

Setup. All experiments are conducted on a machine equipped with an Intel(R) Core(TM) i7-13700K CPU (16 physical cores), 64GB RAM, and an NVIDIA GeForce RTX 4090 GPU with 24GB of memory. We use the latest open-source version of OceanBase [5] in our experiments.

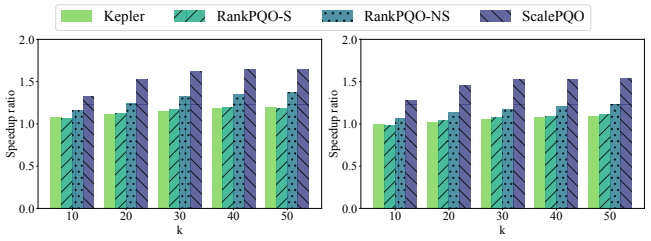
6.2 Overall Performance

Varying dataset. Figure 4 reports the speedup ratios of ScalePQO and baseline methods compared to the OceanBase optimizer, with all other system parameters kept at their default settings. The evaluation covers six workloads, including one anonymized production workload (Anonymized), four synthetic workloads constructed from JOB and DSB under different template scales, and one public StackExchange workload (Stack).

We make the following observations. First, ScalePQO consistently achieves the highest speedup across all evaluated workloads, demonstrating its effectiveness under anonymized production, synthetic benchmark, and public real-world settings. In particular, ScalePQO achieves clear improvements over the strongest baseline RankPQO-NS on all workloads, showing that the proposed clustering and online adaptation strategies are effective beyond the synthetic settings. Second, on the four synthetic workloads, ScalePQO maintains strong performance as the number of query templates increases. For example, when the workload scale increases from JOB330 to JOB3300 and from DSB150 to DSB1500, ScalePQO shows only moderate performance degradation, indicating its scalability under large template sets. Third, the results on Anonymized and Stack further validate the practicality of ScalePQO. On Anonymized, ScalePQO improves performance on a real anonymized production workload. On Stack, ScalePQO also achieves the best speedup on public user-written query templates with naturally collected parameter bindings. These results indicate that the benefits of ScalePQO are not limited to manually constructed synthetic workloads.

Varying the number of cached plans k. Figure 5 shows the impact of varying the number of cached plans k on ScalePQO and baselines. Since OptOnce and Density do not explicitly control the number of cached plans, we exclude them from this experiment. Due to space limitations, we report the results on JOB-based workloads, while similar trends on DSB-based workloads are provided in [28].

Increasing k generally improves all methods by enlarging the candidate plan set. For ScalePQO, the performance improves clearly



(a) JOB330 (b) JOB3300

Figure 5: Varying the number of cached plans k

Table 3: Per group speedup over JOB330

Group	OptOnce	Density	Kepler	-S	-NS	ScalePQO
0	0.99	1.12	1.42	1.53	**1.78**	1.69
1	0.97	1.13	1.29	1.32	1.57	**1.6**
2	0.95	1.14	1.22	1.22	1.37	**1.61**
3	0.95	1.13	1.14	1.17	1.31	**1.62**
4	0.94	1.14	1.11	1.12	1.25	**1.53**
5	0.93	1.12	1.07	1.1	1.21	**1.64**
6	0.94	1.11	1.04	1.09	1.19	**1.61**
7	0.92	1.1	1.06	1.1	1.16	**1.63**
8	0.91	1.09	1.08	1.07	1.18	**1.66**
9	0.9	1.13	1.06	1.07	1.17	**1.58**

when k increases from a small cache size to a moderate one. However, the benefit gradually saturates beyond $k = 30$, suggesting that most useful plans have already been covered, while additional cached plans become increasingly redundant. At $k = 30$, ScalePQO consistently maintains a clear advantage over RankPQO-NS and other baselines under the same cache budget. Therefore, following RankPQO [29], we adopt $k = 30$ as the default setting in our experiments to balance performance and cache overhead.

Per-group performance. Table 3 reports per-group speedup, where each group corresponds to one time slot of the workload. In the table, -S and -NS denote RankPQO-S and RankPQO-NS, respectively. The results show that ScalePQO maintains stable and strong performance across groups. RankPQO-NS performs best in the first group, where the parameter distribution is still close to the training distribution, but its speedup gradually decreases as the workload evolves. Similar degradation can be observed for RankPQO-S and Kepler. In contrast, ScalePQO achieves the best performance in most later groups, showing that online fine-tuning and candidate plan updating help adapt to parameter distribution drift. Density remains relatively stable but provides lower overall speedup than ScalePQO.

Tail latency analysis. Table 4 reports the P90, P95, and P99 tail-latency speedups on JOB330, computed as the corresponding percentile latency of the OceanBase optimizer divided by that of each method. For compactness, -S and -NS denote RankPQO-S and RankPQO-NS, respectively. ScalePQO achieves the highest speedup at all percentiles, reaching 1.70×, 1.77×, and 1.93× at P90, P95, and P99. The larger gains at higher percentiles indicate that ScalePQO is particularly effective in reducing slow query instances, which is important for industrial workloads with strict latency requirements.

6.3 Impact of Template Clustering Strategies

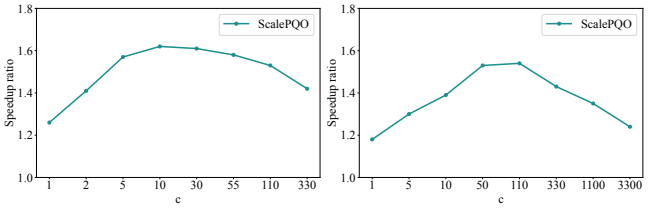
Clustering strategy comparison. Table 5 compares edit-distance-based clustering with our hierarchical-model-based clustering strategy. The former groups query templates by syntactic similarity,

Table 4: Tail latency speedup over JOB330

	OptOnce	Density	Kepler	-S	-NS	ScalePQO
P90	0.95	1.18	1.19	1.2	1.43	**1.7**
P95	0.96	1.24	1.27	1.37	1.48	**1.77**
P99	0.99	1.29	1.4	1.47	1.65	**1.93**

Table 5: Clustering strategy comparison

	Edit-distance-based	Hierarchical-model-based
JOB330	1.4	1.62
JOB3300	1.29	1.53
DSB150	1.39	1.6
DSB1500	1.27	1.49



(a) JOB330 (b) JOB3300

Figure 6: Varying the number of clusters c

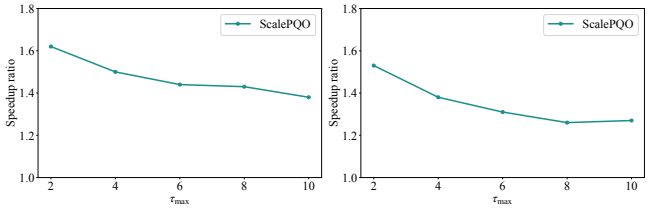
while the latter uses learned template embeddings to guide clustering and model fine-tuning. Hierarchical-model-based clustering consistently achieves higher speedups across all workloads, improving from 1.40 to 1.62 on JOB330, 1.29 to 1.53 on JOB3300, 1.39 to 1.60 on DSB150, and 1.27 to 1.49 on DSB1500. This demonstrates that learned embeddings better capture performance-relevant similarities among templates than surface-level syntax, leading to more effective model specialization and knowledge sharing.

Varying the number of clusters c. Figure 6 presents the impact of varying the number of clusters c in our hierarchical training framework. Due to space limitations, we report the results on JOB-based workloads, with more details provided in [28]. Overall, a larger c improves model specialization at first, but the benefit saturates or may decline when training data becomes fragmented. Based on these results, we adopt $c = 10$ for JOB330 and $c = 50$ for JOB3300 as the default settings.

6.4 Impact of Online Fine-Tuning Strategy

Varying the threshold for full fine-tuning $\tau_{\max}$. Figure 7 studies the impact of $\tau_{\max}$, which controls the aggressiveness of online adaptation. A smaller $\tau_{\max}$ triggers more timely fine-tuning and plan re-selection, while a larger value delays updates. Due to space limitations, we report JOB-based results here, with more details provided in [28]. Overall, performance decreases as $\tau_{\max}$ increases, because overly conservative adaptation may keep stale models and outdated cached plans for longer. Based on these results, we adopt $\tau_{\max} = 2$ as the default setting to balance adaptation effectiveness and update frequency.

Plan updating policy comparison. Table 6 compares two cached-plan update policies: *Random*, which updates arbitrary cache positions, and *Prepend*, which inserts newly selected plans at the front and evicts the oldest ones. *Prepend* consistently outperforms *Random* across all workloads, showing that preserving temporal locality



(a) JOB330 **(b) JOB3300**

Figure 7: Varying the threshold for full fine-tuning $\tau_{\max}$

Table 6: Plan updating policy comparison

	Random	Prepend
JOB330	1.43	1.62
JOB3300	1.34	1.53
DSB150	1.41	1.6
DSB1500	1.3	1.49

helps the cache better follow workload shifts and avoid stale plans. More details are provided in [28].

6.5 Overhead Analysis

We evaluate the efficiency of ScalePQO by reporting detailed offline and online overheads on the JOB330 workload. This workload contains 330 query templates and is divided into 10 time slots, with each slot containing 5 parameter vectors per template, resulting in a total of 16,500 queries. In Table 7, "ScalePQO-E" refers to the variant that uses *edit-distance-based clustering*, while "ScalePQO" adopts the *hierarchical-model-based clustering* strategy. All other system parameters follow the default settings described earlier. Since OptOnce and Density do not involve model training or multi-stage plan selection pipelines, we exclude them from the pipeline efficiency breakdown.

Offline overhead. The offline stage includes plan generation, template clustering, model training, and candidate plan selection. As shown in Table 7, the main overhead differences come from clustering and model training. ScalePQO significantly reduces clustering cost compared with ScalePQO-E (68s vs. 479s) by using embedding-based clustering instead of pairwise edit-distance computation. For model training, RankPQO-NS has the highest cost because it trains a separate model for each template, while ScalePQO uses global training followed by cluster-level fine-tuning, reducing the training time from 5868s to 3826s. All offline operations are one-time costs and do not affect online query latency. More details are provided in [28].

Online overhead. During query processing, the dominant component across all methods is plan execution, which accounts for the majority of runtime. Compared with Kepler, ScalePQO spends slightly more time on best plan selection (e.g., 446s vs. 285s) but is comparable to the RankPQO variants, and it substantially reduces the plan execution time, yielding the shortest overall online processing time. Two additional operations, model fine-tuning and candidate plan updating, are unique to ScalePQO and ScalePQO-E. Although these contribute extra work, they run concurrently with query execution and therefore do not extend the end-to-end query latency. As a result, ScalePQO achieves superior overall performance without increasing the total online processing time.

Training overhead. We profile the training overhead using a 5,000-template workload generated from JOB, which approximates the largest production-scale workload in Table 1. Table 8 reports the wall-clock training time and model size, with more details provided in [28]. Since we fix the total number of training samples following Section 3.1, training time is mainly determined by the sample budget and model architecture rather than growing linearly with the number of templates. Compared with RankPQO-NS, ScalePQO significantly reduces training and storage overhead while achieving better performance.

6.6 Limitations

While ScalePQO demonstrates strong performance in industrial settings, it still has several limitations. First, our approach focuses on parameter distribution drift and assumes a relatively stable data distribution. In practice, changes in underlying data statistics (e.g., data skew or updates) may also affect query performance, which is not explicitly handled in our current framework.

Second, the effectiveness of template clustering depends on the quality of learned embeddings, the aggregation strategy used to construct template-level representations, and the choice of clustering algorithm. If the embedding model fails to accurately capture performance-related similarities between templates, or if the aggregation strategy loses important distributional information, the resulting clusters may be suboptimal and affect model specialization. In particular, our current template embedding uses mean pooling over parameter and plan embeddings, which provides a lightweight summary for scalable clustering but may not fully capture highly multimodal parameter distributions within a template. In addition, we use k-means as a practical and scalable partitioning method, but we have not systematically explored alternative clustering algorithms, such as DBSCAN, agglomerative clustering, or spectral clustering. Investigating more expressive template representations, such as multi-prototype or distribution-aware embeddings, together with alternative clustering algorithms is an interesting direction for future work.

Third, ScalePQO relies on candidate plan enumeration strategies adapted from prior work. The quality of the final plan selection is therefore bounded by the diversity and coverage of the generated candidate plans. Finally, although our online adaptation mechanism is designed to be lightweight and executed asynchronously, it still introduces additional computational overhead. In resource-constrained environments, this overhead may need to be carefully managed. Addressing these limitations is part of our future work toward building more robust and fully adaptive PQO systems.

7 RELATED WORK

Learned Parametric Query Optimization. Recent work on PQO [10, 23, 24, 29, 31] has leveraged machine learning to improve plan selection, following a two-stage framework: (1) candidate plan enumeration and (2) best plan selection. For candidate plan enumeration, query-log-driven approaches [31] rely on previously executed plans, limiting adaptability, while heuristic methods like Kepler [10] adjust cardinality estimates but face efficiency issues. RankPQO [29] improves plan diversity by jointly modifying cardinality and join

Table 7: Efficiency of pipeline. Times in seconds.

Stage	Breakdown Item	Kepler	RankPQO-S	RankPQO-NS	ScalePQO-E	ScalePQO
Offline	Plan generation	1193	3285	3285	3285	3285
	Template clustering	—	—	—	479	68
	Model training	564	1545	5868	2161	3826
	Candidate plan selection	126	1715	1734	1698	1709
Online	Best plan selection	285	432	473	453	446
	Model finetuning	—	—	—	798	810
	Candidate plan updating	—	—	—	357	342
	Plan Execution	25545	25093	22233	20961	18110

Table 8: Training overhead with 5,000 query templates.

	RankPQO-S	RankPQO-NS	ScalePQO
Training time (s)	1753	6194	4225
Model size (MB)	744	5712	759

orders. For best plan selection, classification and regression models [10, 31] assume smooth parameter-plan relationships, which fail in cases of abrupt plan transitions. RankPQO [29] addresses this by adopting a learning-to-rank model that focuses on relative plan performance. While these methods show promising results, they overlook industrial challenges such as large-scale query templates and evolving parameter distributions. Our work extends learned PQO techniques to address these issues (see Section 3.2).

RankPQO. Mo et al. [29] proposed RankPQO, a learning-to-rank approach for PQO that addresses plan enumeration and selection jointly. For plan enumeration, RankPQO introduces a hybrid strategy that simultaneously modifies cardinality estimates and join orders, increasing plan diversity and improving plan quality. For plan selection, it employs a ranking model that predicts relative plan performance instead of relying on absolute cost estimations, enhancing robustness against parameter distribution shifts. While RankPQO achieves strong performance in experimental settings, it does not account for scalability and adaptation challenges in industrial workloads.

OceanBase. OceanBase [34] is a distributed relational database system widely used in financial and e-commerce applications. It supports high-throughput transactional workloads and provides low-latency query execution through a plan caching mechanism, which reuses execution plans for repeated queries to reduce optimization overhead. For parametric queries, OceanBase caches execution plans based on observed parameter values, allowing efficient reuse. However, as workloads evolve, the cached plan set continues to grow, leading to increased memory usage and potential suboptimal plan selections. This motivates the need for an adaptive mechanism to refine and maintain an optimal set of cached plans.

ML for Query Optimization. Recent work has applied machine learning to various aspects of query optimization [20, 36], including cardinality estimation [13, 18, 30] and end-to-end query optimizers [25–27, 33, 35, 37]. Among them, systems like Neo [26] and Balsa [33] learn to generate plans from scratch, while Bao [25], HybridQO [35], and Lero [37] leverage learned models for plan selection over candidates from traditional optimizers. Our work differs by targeting parametric query optimization, where the goal is to cache a small number of plans per template and select the best one for varying user parameters. This requires modeling execution cost as a function of both the plan and input parameters, which existing query optimizers do not support. Other works, such as CARDLEARNER [32] and Sibyl [14], explore template-based cost estimation and time-evolving workload forecasting, respectively. These objectives differ fundamentally from ScalePQO, which targets parametric query optimization and adaptive plan caching.

8 LESSONS LEARNED

Based on our experience of designing and deploying ScalePQO in an industrial database system, we summarize several key lessons.

First, there exists a fundamental trade-off between model scalability and prediction accuracy in PQO. While per-template models achieve strong performance, they are not scalable in large workloads. In contrast, a single shared model scales well but suffers from significant performance degradation. Our results show that clustering-based model sharing provides an effective balance between these two extremes. Second, clustering based on learned representations is more effective than relying on query syntax alone. Traditional edit-distance-based methods fail to capture performance-relevant similarities, whereas embedding-based clustering better aligns with optimization behavior and leads to improved plan selection. Third, parameter distribution drift is inevitable in real-world workloads and has a significant impact on PQO performance. Static models and fixed plan caches degrade over time, highlighting the necessity of adaptive mechanisms in industrial systems. Finally, the design of online adaptation must carefully balance effectiveness and overhead. Lightweight and asynchronous model fine-tuning and plan updating are essential to ensure that adaptation does not introduce additional latency to query execution.

These observations provide practical guidance for designing scalable and adaptive PQO systems in real-world deployments.

9 CONCLUSION

We presented ScalePQO, a scalable and adaptive framework for Parametric Query Optimization (PQO) in industrial settings. By combining representation-based template clustering with hierarchical model training, and introducing a KL-divergence-driven fine-tuning and plan re-selection strategy, ScalePQO effectively addresses scalability and adaptability challenges in real-world workloads. Integrated into OceanBase, ScalePQO achieves up to 1.62× speedup over the OceanBase optimizer and outperforms state-of-the-art methods like RankPQO, demonstrating its practical efficiency and robustness.

ACKNOWLEDGMENTS

This research is supported in part by Singapore MOE AcRF Tier-2 grant MOE-T2EP20223-0004 and a grant from OceanBase.

REFERENCES

[1] [n.d.]. OceanBase Plan Cache. https://en.oceanbase.com/docs/common-oceanbase-database-10000000001123504.

[2] [n.d.]. SQL Server Plan Cache Object. https://docs.microsoft.com/en-us/sql/relational-databases/performance-monitor/sql-server-plan-cache-object?view=sql-server-ver15.

[3] [n.d.]. StackExchange. https://data.stackexchange.com/stackoverflow/queries.

[4] 2024. Postgres hint plan. https://github.com/ossc-db/pg_hint_plan.

[5] January 2025. OceanBase. https://github.com/oceanbase/oceanbase.

[6] Günes Aluç, David DeHaan, and Ivan T. Bowman. 2012. Parametric Plan Caching Using Density-Based Clustering. In *IEEE 28th International Conference on Data Engineering (ICDE 2012), Washington, DC, USA (Arlington, Virginia), 1-5 April, 2012*. IEEE Computer Society, 402–413.

[7] Pedro Bizarro, Nicolas Bruno, and David J. DeWitt. 2009. Progressive Parametric Query Optimization. *IEEE Trans. Knowl. Data Eng.* 21, 4 (2009), 582–594.

[8] Surajit Chaudhuri, Hongrae Lee, and Vivek R. Narasayya. 2010. Variance aware optimization of parameterized queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*. ACM, 531–542.

[9] Bailu Ding, Surajit Chaudhuri, Johannes Gehrke, and Vivek R. Narasayya. 2021. DSB: A Decision Support Benchmark for Workload-Driven and Traditional Database Systems. *Proc. VLDB Endow.* 14, 13 (2021), 3376–3388.

[10] Lyric Doshi, Vincent Zhuang, Gaurav Jain, Ryan Marcus, Haoyu Huang, Deniz Altinbüken, Eugene Brevdo, and Campbell Fraser. 2023. Kepler: Robust Learning for Parametric Query Optimization. *Proc. ACM Manag. Data* 1, 1 (2023), 109:1–109:25.

[11] Anshuman Dutt, Vivek R. Narasayya, and Surajit Chaudhuri. 2017. Leveraging Re-costing for Online Optimization of Parameterized Queries with Guarantees. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*. ACM, 1539–1554.

[12] Sumit Ganguly. 1998. Design and Analysis of Parametric Query Optimization Algorithms. In *VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, August 24-27, 1998, New York City, New York, USA*. Morgan Kaufmann, 228–238.

[13] Yuxing Han, Ziniu Wu, Peizhi Wu, Rong Zhu, Jingyi Yang, Liang Wei Tan, Kai Zeng, Gao Cong, Yanzhao Qin, Andreas Pfadler, Zhengping Qian, Jingren Zhou, Jiangneng Li, and Bin Cui. 2021. Cardinality Estimation in DBMS: A Comprehensive Benchmark Evaluation. *Proc. VLDB Endow.* 15, 4 (2021), 752–765.

[14] Hanxian Huang, Tarique Siddiqui, Rana Alotaibi, Carlo Curino, Jyoti Leeka, Alekh Jindal, Jishen Zhao, Jesús Camacho-Rodríguez, and Yuanyuan Tian. 2024. Sibyl: Forecasting Time-Evolving Query Workloads. *Proc. ACM Manag. Data* 2, 1 (2024), 53:1–53:27.

[15] Arvind Hulgeri and S. Sudarshan. 2002. Parametric Query Optimization for Linear and Piecewise Linear Cost Functions. In *Proceedings of 28th International Conference on Very Large Data Bases, VLDB 2002, Hong Kong, August 20-23, 2002*. Morgan Kaufmann, 167–178.

[16] Arvind Hulgeri and S. Sudarshan. 2003. AniPQO: Almost Non-intrusive Parametric Query Optimization for Nonlinear Cost Functions. In *Proceedings of 29th International Conference on Very Large Data Bases, VLDB 2003, Berlin, Germany, September 9-12, 2003*. Morgan Kaufmann, 766–777.

[17] Yannis E. Ioannidis, Raymond T. Ng, Kyuseok Shim, and Timos K. Sellis. 1997. Parametric Query Optimization. *VLDB J.* 6, 2 (1997), 132–151.

[18] Kyoungmin Kim, Jisung Jung, In Seo, Wook-Shin Han, Kangwoo Choi, and Jaehyok Chong. 2022. Learned Cardinality Estimation: An In-depth Study. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*. ACM, 1214–1227.

[19] Gökhan Kul, Duc Thanh Anh Luong, Ting Xie, Varun Chandola, Oliver Kennedy, and Shambhu J. Upadhyaya. 2018. Similarity Metrics for SQL Query Clustering. *IEEE Trans. Knowl. Data Eng.* 30, 12 (2018), 2408–2420.

[20] Hai Lan, Zhifeng Bao, and Yuwei Peng. 2021. A Survey on Advancing the DBMS Query Optimizer: Cardinality Estimation, Cost Model, and Plan Enumeration. *Data Sci. Eng.* 6, 1 (2021), 86–101.

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

[22] Vladimir I Levenshtein et al. 1966. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, Vol. 10. Soviet Union, 707–710.

[23] Sijia Li, Peng Cai, Yiqi Shen, Huiqi Hu, Rong Zhang, Xuan Zhou, Quanqing Xu, and Ri Zhao. 2024. SPQO: Learning to Safely Reuse Cached Plans for Dynamic Workloads. In *International Conference on Database Systems for Advanced Applications (DASFAA) (Lecture Notes in Computer Science)*, Vol. 14850. Springer, 315–330.

[24] Sijia Li, Peng Cai, Zhifan Zhang, Huiqi Hu, Rong Zhang, Xuan Zhou, Quanqing Xu, and Chuanhui Yang. 2025. APQO: An Adaptive Framework for Parametric Query Optimization. *Proceedings of the ACM on Management of Data* 3, 6, Article 296 (2025), 25 pages.

[25] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2021. Bao: Making Learned Query Optimization Practical. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*. ACM, 1275–1288.

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

[27] Songsong Mo, Yile Chen, Hao Wang, Gao Cong, and Zhifeng Bao. 2023. Lemo: A Cache-Enhanced Learned Optimizer for Concurrent Queries. *Proc. ACM Manag. Data* 1, 4 (2023), 247:1–247:26.

[28] Songsong Mo, Quanqing Xu, Xuchen Ding, Yue Zhao, Zhifeng Bao, Chuanhui Yang, and Gao Cong. 2026. Towards Industrial-Scale Parametric Query Optimization [Technical Report]. (2026), 1–15. https://github.com/songsong945/RankPQO-industry/blob/main/ScalePQO_Technical_Report.pdf

[29] Songsong Mo, Yue Zhao, Zhifeng Bao, Quanqing Xu, Chuanhui Yang, and Gao Cong. 2024. RankPQO: Learning-to-Rank for Parametric Query Optimization. *Proc. VLDB Endow.* 18, 3 (2024), 863–875.

[30] Ji Sun, Jintao Zhang, Zhaoyan Sun, Guoliang Li, and Nan Tang. 2021. Learned Cardinality Estimation: A Design Space Exploration and A Comparative Evaluation. *Proc. VLDB Endow.* 15, 1 (2021), 85–97.

[31] Kapil Vaidya, Anshuman Dutt, Vivek R. Narasayya, and Surajit Chaudhuri. 2021. Leveraging Query Logs and Machine Learning for Parametric Query Optimization. *Proc. VLDB Endow.* 15, 3 (2021), 401–413.

[32] Chenggang Wu, Alekh Jindal, Saeed Amizadeh, Hiren Patel, Wangchao Le, Shi Qiao, and Sriram Rao. 2018. Towards a Learning Optimizer for Shared Clouds. *Proc. VLDB Endow.* 12, 3 (2018), 210–222.

[33] 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 ACM SIGMOD International Conference on Management of Data, SIGMOD 2022, Philadelphia, PA, USA, June 12 - 17, 2022*. ACM, 931–944.

[34] Zhenkun Yang, Chuanhui Yang, Fusheng Han, Mingqiang Zhuang, Bing Yang, Zhifeng Yang, Xiaojun Cheng, Yuzhong Zhao, Wenhui Shi, Huafeng Xi, Huang Yu, Bin Liu, Yi Pan, Boxue Yin, Junquan Chen, and Quanqing Xu. 2022. OceanBase: A 707 Million tpmC Distributed Relational Database System. *Proc. VLDB Endow.* 15, 12 (2022), 3385–3397.

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

[36] Yue Zhao, Songsong Mo, and Gao Cong. 2025. TATA: An Efficient Framework for Task Transfer in Query Plan Representation. *Proc. VLDB Endow.* 19, 3 (2025), 413–425.

[37] Rong Zhu, Wei Chen, Bolin Ding, Xingguang Chen, Andreas Pfadler, Ziniu Wu, and Jingren Zhou. 2023. Lero: A Learning-to-Rank Query Optimizer. *Proc. VLDB Endow.* 16, 6 (2023), 1466–1479.